Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Scares For Sale: Diseño y desarrollo de un videojuego en Unity3D. Audio e introducción a LeapMotion

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Sergio Alapont Granero

Tutor: Javier Lluch Crespo

2013/2014

Resumen

Este proyecto desarrolla la implementación de los sistemas de entrada y de audio empleados dentro del videojuego *Scares For Sale*. Además incorpora mecánicas innovadoras con el uso del dispositivo *LeapMotion* y de librerías de reconocimiento de formas para ofrecer al usuario experiencias únicas en la jugabilidad.

Palabras clave: Videojuego, Entrada, Audio, Markov, LeapMotion, Unity3D

Abstract

This project develops the audio and input system implementation used inside the videogame *Scares For Sale*. It also incorporates new features by using the *LeapMotion* device and *Pattern Recognition* libraries to provide users with new gameplay experiences.

Keywords : Videogame, Input, Audio, Markov, LeapMotion, Unity3D

Agradecimientos

A mi familia por darme todo su apoyo durante toda mi vida, así como su paciencia y sus consejos.

A mis amigos y a mis compañeros por estar ahí en todo momento durante esta etapa de mi vida.

A Javier Lluch Crespo por supervisar el desarrollo de este proyecto.

A todos los miembros de BraveZebra por luchar día tras día en convertirnos en un estudio grande de desarrollo de videojuegos.

Tabla de contenidos

Índice de imágenes

Índice de tablas

1. Introducción

El proyecto que se describe en esta memoria trata sobre la realización de un videojuego a partir de los diferentes elementos que lo componen, diseño, programación, integración de modelos en escena y diseño e implementación de interfaces utilizando para ello un motor profesional de videojuegos, *Unity*.

El videojuego será del género *Tower Defense*. En el juego el jugador encarnará a un fantasma cuyo objetivo es proteger una mansión de los compradores que la visitan, si un cierto número de estos compradores llega a hacer un recorrido completo hasta el tejado de la mansión, tendrá lugar la subasta de la casa y por lo tanto el fin del juego.

Para ahuyentar a las oleadas de compradores de la mansión, controlados por la IA del juego, el usuario deberá utilizar los sustos o habilidades que poseen los objetos encantados de la mansión, como pueden ser, las cortinas, sillas, bustos, armaduras, lámparas, etc. Cada uno de estos elementos encantados poseerá una habilidad única con tres niveles de efectividad, dichos niveles deberán adquirirse por el jugador con el dinero que dejan los visitantes al huir despavoridos tras superar un nivel.

El contenido presentado en esta memoria agrupa una de las tres partes del total del videojuego *Scares For Sale* que han sido supervisadas bajo la atención del profesor Javier Lluch Crespo. Las partes constituyentes del total son las siguientes.
La primera parte (*Diseño y desarrollo de un videojuego 3D sobre Unity : Inteligencia artificial)* ha sido desarrollada por David Fernández Molina. Este trabajo desarrolla una extensión de Unity con el fin de poder gestionar máquinas de estados finitas para las inteligencias artificiales en videojuegos.
La segunda parte (*Diseño y desarrollo de un videojuego 3D sobre Unity : Interfaces y trampas*) ha sido desarrollada por Agustín Esteve Guinot. Este proyecto desarrolla una extensión de Unity para la elaboración de interfaces de usuario usando el *plugin NGUI*. Además, aquí se elaboran las trampas empleadas dentro del videojuego elaborado.

Por último, la parte aquí desarrollada conlleva la gestión de los controles de juego, así como el audio del mismo. Para ello, dada la simplicidad ofrecida por *Unity* se ha realizado una serie de sistemas con el fin de darle robustez y potencia al usuario. Por un lado, la primera extensión conlleva un gestor completo de audio basado en canales con el que controlar los sonidos y la música de nuestra aplicación de una forma rápida y sencilla para el usuario. Por otro lado, dado el bajo nivel de las librerías propias de *Unity* para la gestión de los controles de juego, se ha creado un sistema para llevar la gestión de las mecánicas de entrada más comunes en los videojuegos de hoy día para plataformas móviles y para ordenador.
Además, dada la simplicidad de los *plugins* orientados a la entrada que pueden encontrarse en la *Asset Store*, se decidió innovar para este proyecto incluyendo una serie de mecánicas novedosas. En primer lugar se incluyó la posibilidad de controlar una aplicación sin necesidad de ningún periférico de entrada convencional (ratón, teclado) en la plataforma *PC* gracias al dispositivo *LeapMotion* desarrollado en el año 2013. Por otro lado, tomando como referencia el reconocimiento de formas de los videojuegos *Brain Training*, se decidió incluir para nuestro sistema un reconocedor de

trazos empleando para ello librerías de modelos de *Markov* contenidas dentro del *framework Accord.NET*.

La estructura del proyecto comienza en la sección *Estado del Arte* con una presentación de los distintos entornos de desarrollo que existen en la industria, así como sus características. Además, se realiza una comparativa de los tres lenguajes empleados en *Unity*.
A continuación, en la sección *Antecedentes* se presentan las tecnologías empleadas con una breve explicación de las mismas. Esta explicación comienza con el entorno empleado y continua con el audio dentro de *Unity3D* y la presentación del *SDK* del dispositivo *LeapMotion* y las librerías *Accord.NET*.

El siguiente paso, en la sección *Análisis y Diseño*, continua con el planteamiento de los distintos problemas a resolver en los sistemas a realizar y la solución planteada para ellos.

Una vez se han planteado los campos del proyecto con sus correspondientes funcionalidades se realiza la explicación de la implementación de los mismos en el apartado *Implementación* donde se detalla que funcionalidades de las librerías empleadas se han utilizado en el proceso de codificación.

Por último, se ha realizado una revisión de los resultados obtenidos, a posteriori, comprobando el correcto funcionamiento y la usabilidad de la resolución a los objetivos planteados. Esto se realiza en la sección *Resultados*.

2. Objetivos

Cuando un desarrollador se enfrenta al desafío de implementar un videojuego con Unity hay ciertos factores que pueden entorpecer el proceso si no se conoce bien el correcto uso de las herramientas proporcionadas por el entorno. Si además se planea desarrollar un número abundante de videojuegos es una idea lógica la de desarrollar un sistema propio que haga uso de estas herramientas y agilice el proceso de desarrollo.

En la fase de diseño del videojuego *Scares For Sale* se decidió crear un sistema de audio con el fin de usarse posteriormente en futuros proyectos.
Para esta arquitectura se decidió la siguiente lista de funcionalidades.

- Proporcionar al usuario la capacidad de gestionar el audio en una serie de canales de forma separada.
- Gestión de la dispersión del audio de forma que aquellos canales más prioritarios sean los que posean un nivel auditivo más elevado.
- Decidir, dada una pista de audio, el intervalo de tiempo a escuchar cuando el usuario la ejecuta.
- Dotar para cada pista de audio un sistema de eventos de forma que puedan emplearse como lanzadores de otras acciones internas del juego.
- El sistema proporcionará información visual acerca del estado de cada emisor así como del rango de alcance del audio que podrá modificarse por el usuario desde el propio editor.

Por otro lado se ideó un sistema de entrada para poder realizar una gestión cómoda y abstraída de los controles del juego de forma que la realización de código destinada para este fin en futuros proyectos fuera mínima. Las funcionalidades pensadas para la versión de este sistema en la realización de este proyecto son las siguientes.

- Gestión de *swipes* y de pulsaciones tanto en plataformas móviles como para ordenador.
- Dotar al usuario de una serie de parámetros usados para la personalización del comportamiento de estos gestos.
- Proporcionar basado en modelos ocultos de *Markov* donde el usuario se podrá definir sus propios trazos a reconocer posteriormente dentro de la aplicación.
- Reconocer dentro del *gameplay* si se ha realizado un trazo de entre los anteriormente definidos.

Además, se ha considerado oportuno incorporar a la arquitectura la posibilidad de emplear el *Leap Motion* para proporcionar al usuario la capacidad de explotar las funcionalidades previamente descritas, en ordenador, sin necesidad de emplear el uso del teclado o del ratón.

Por último, se debía realizar un sistema de movimiento de cámara que permitiera al jugador desplazarse en el escenario del juego con las funcionalidades siguientes.

- Desplazamiento vertical entre las distintas plantas del edificio.
- Rotación entre las distintas caras del edificio.

– Animación (*Fade in* y *Fade out*) de la transparencia los objetos (paredes) que dificulten la visibilidad.

3. Estado del arte

Durante la última década, con la aparición de los *smartphones*, se ha popularizado el género *casual* en el mercado de los videojuegos convirtiendo este sector en el más rentable dentro de la industria del entretenimiento. A esto se le ha añadido la aproximación de los distintos entornos de desarrollo a los creadores permitiendo un incremento tanto en la variedad como en el número de aplicaciones en los mercados.

En particular, se decidió realizar el juego *Scares For Sale* tomando como referencias algunos videojuegos de éxito del género *Tower Defense*. Estas referencias corresponden a: *Plants Vs Zombies* (2009), *Orcs Must Die* (2011) y *Luigi's Mansión: Dark Moon* (2013).

3.1 Comparativa con otros motores

Los entornos de desarrollo más empleados en la actualidad se agrupan en *Unreal Engine*, *Unity3D* y *Cry Engine*. A la hora de plantearse el desarrollo de un videojuego es necesario decidir si se va a realizar desde la nada o, por el contrario, se va a emplear un sistema como base para la implementación. En este último caso el usuario se enfrenta a la toma de una segunda decisión, la elección de dicho sistema. Para ello se debe conocer cuál es el abanico de posibilidades al alcance y escoger aquel cuyas prestaciones más se adecuen a las necesidades del juego. Por norma general todos los motores suelen ofrecer las mismas herramientas con mayor o menor rendimiento, en cuyo caso el usuario a de escoger aquel cuya usabilidad le sea más cómoda.

3.1.1 Unreal Engine

Unreal Engine **[1]** es, sin duda, el motor de videojuegos 3D más extendido en cuanto a videojuegos profesionales se refiere. Su desarrollo está a cargo de la compañía Epic Games, creadores de juegos como Gears of War, quienes comenzaron el desarrollo del motor en 1998 aplicando en él las más novedosas técnicas de los videojuegos hasta llegar a su versión actual: Unreal Engine 4.

Unreal Engine solía ser el motor elegido por la mayoría de desarrolladores debido a la fácil accesibilidad del motor, pues constaba de un kit de desarrollo totalmente gratuito (con ciertas limitaciones) con numerosos modelados, sistemas de partículas, scripts e incluso proyectos ya terminados con los que aprender a utilizar el motor.

Hasta la versión cuatro, los usuarios de UDK podían utilizar el kit de manera gratuita hasta comenzar a comercializar el juego, en este caso los desarrolladores debían pagar 100 dólares y un 25% del total de ganancias después de conseguir unos beneficios totales de 50.000 dólares. Si los desarrolladores poseían cierto renombre podían

contactar con Epic Games para adquirir total acceso al motor UE3 negociando el precio.

Todo ello cambió con la llegada de UE4, el nuevo modelo de negocio de Epic Games, pasa a cobrar 20 dólares al mes más un 5% de las ventas, para tener acceso Al código de la última versión de Unreal Engine directamente desde el repositorio en GITHUB, teniendo que compilar los propios usuarios el proyecto desde Visual Studio.

A continuación se muestra una comparativa resumen sobre los pros y contras que Unreal Engine presenta sobre Unity:

	Unreal	Unity
Pros	Al pagar una mensualidad ya se posee una versión (no actualizada) del proyecto. Acceso al código fuente del motor (C++). Resultados visuales más aparentes que con otros motores debido sobre todo a los efectos de iluminación. Creación de materiales, *shaders* incluso scripts desde editores gráficos (con grafos).	Versión gratuita, ampliable mediante plugins, suficiente para la mayoría de proyectos. Posibilidad de programar con lenguajes más conocidos y con gestión dinámica de memoria (C# y Javascript). Shaders programados mediante ShaderLab, un lenguaje muy similar a Cg y GLSL.
Contras	Imposibilidad de probar el motor sin pagar. Comunidad pequeña con pocos tutoriales y facilidades para aprender en comparación con otros motores. Imposibilidad de publicar un juego si no se ha pagado la mensualidad requerida.	Versión Pro demasiado cara para pequeños desarrolladores. Falta de comodidades para los no programadores.

Tabla 1: Comparativa. Unreal VS Unity3D

3.1.2 CryEngine

CryEngine **[2]** se ha mostrado desde 2002 como uno de los motores más potentes de la historia, superando en varios aspectos a *UnrealEngine*.

Diseñado por *Crytek*, *CryEngine* no se ha extendido entre los desarrolladores de manera tan amplia como si lo han hecho *Unreal* o *Unity*, pero los pocos juegos en los que ha sido utilizado han hecho que el motor se gane su fama, sobre todo por su gran potencia gráfica.

Algunos de los juegos desarrollados con *CryEngine* son: Crysis (desarrollado por la propia *Crytek*), Los Sims 3, Aion Online, Monster Hunter Online o la franquicia Far Cry.

De la misma forma que *Unreal*, *CryEngine* cambió su modelo de negocio en 2014 para pasar a cobrar 9.90€ al mes por obtener la última versión del motor, recuperando parte del terreno que tenía perdido ante *Unity* y *Unreal* por su exclusividad.

A continuación se muestra una comparativa resumen sobre los pros y contras que *Cry Engine* presenta sobre *Unity*:

	CryEngine	Unity
Pros	Al pagar una mensualidad ya se posee una versión del proyecto. Gran acabado visual de los videojuegos sin necesidad de añadidos. Creación de scripts de juego (eventos, reglas de juego, etc) con editor de flujo gráficos. Uso del lenguaje LUA para describir reglas de juego, Inteligencia artificial, comunicación en red, si no se quiere utilizar los editor gráficos, aunque la programación general se realiza en C++.	Versión gratuita, ampliable mediante plugins, suficiente para la mayoría de proyectos. Soporte para más plataformas, no solo consolas (como Xbox, Wii o PS4) si no también Mac, flash, webgl, iOS, Android y Blackberry. Posibilidad de programar con lenguajes más conocidos y con gestión dinámica de memoria (C# y Javascript). Shaders programados mediante ShaderLab, un lenguaje muy similar a Cg y GLSL.
Contras	Imposibilidad de probar el motor sin pagar. Comunidad muy reducida, foro con pocos usuarios y con escasos tutoriales. No se puede publicar un juego si no se ha pagado la mensualidad requerida.	Versión Pro demasiado cara para pequeños desarrolladores. Falta de comodidades para los no programadores.

Tabla 2: Comparativa. Cry Engine VS Unity3D

3.2 Comparativa de lenguajes empleados en Unity

Cuando se comienza a trabajar empleando *Unity* hay una serie de premisas que deben cumplirse, una de ellas es la de escoger el lenguaje o lenguajes de programación de los que se hará uso a lo largo del proceso de scripting en las aplicaciones pertinentes.
En el caso de *Unity* el usuario puede elegir trabajar con C#, con UnityScript o con Boo, siendo este último el menos utilizado por la comunidad.
Escoger uno de estos tres lenguajes para comenzar a escribir código para *Unity* es un debate que comenzó en los inicios del propio entorno, dado que cada lenguaje proporciona sus propias características.

- **C#**: Gran cantidad de desarrolladores escogen C# para sus aplicaciones, esto se debe al propio origen del lenguaje. Este lenguaje de programación

orientada a objetos fue desarrollado por Microsoft e introducido dentro de la plataforma .NET. Fue creado con influencias de lenguajes basados en C entre los que se encuentran C++ y Java, esto ha desencadenado que muchos desarrolladores provenientes de estos lenguajes escojan C# como el lenguaje a emplear en los scripts usados en *Unity*.

- **UnityScript**: Este lenguaje es a menudo confundido con JavaScript por muchos usuarios entre los que se encuentran miembros de la propia compañía.
  Mientras que JavaScript es un nombre el cual hace referencia a la especificación ECMAScript, UnityScript (.js) es un lenguaje creado por los desarrolladores de Unity para su uso dentro del entorno. Ese es el motivo por el que hay mucha confusión en la red cuando se trata de buscar información fiable acerca de UnityScript.
- **Boo**: Este lenguaje basado en *Python* fue creado con el objetivo de proporcionar al usuario la agilidad propia de *Python* y las potentes características de la plataforma .NET. Se caracteriza por su sintaxis y sistema de identación semejante al de *Python* el cual permite declaración automática de variables, currificación e inferencia de tipos.
  A diferencia de otros lenguajes POO como C# o Java, *Boo* no necesita la creación de una clase para escribir código ejecutable, es decir el uso de clases es algo opcional.

Sin embargo, son C# y *UnityScript* las opciones comúnmente escogidas por la comunidad a la hora de trabajar con los scripts de *Unity* dado que aunque *Boo* es un lenguaje de fácil comprensión y aprendizaje, su influencia en la red no tiene una gran extensión lo que provoca dificultades por parte de los programadores a la hora de buscar documentación y tutoriales útiles para las necesidades que aparecen durante el proceso de desarrollo.

En el caso de *Scares For Sale* se ha escogido C# como lenguaje de programación. El motivo de esta elección radica en la familiaridad con los lenguajes basados en C aprendidos durante la etapa universitaria donde se adquirieron conocimientos de Java, C y C++ entre otros lenguajes. Otro punto a favor de esta elección se debe a que *Unity* está construido sobre la infraestructura .NET siendo el uso de este *framework* algo cotidiano durante la elaboración de los scripts. Además hay que destacar que a la hora de buscar información acerca de las librerías .NET solo se encuentran ejemplos de código en C++, C#, F# y VB.
Por otro lado, toda la documentación y librerías propias de *Unity* aportan el código de sus ejemplos escritos en *UnityScript*, siendo el número de ejemplos escritos en C# más reducido.

C#	
Ventajas	Inconvenientes
- Proviene de .NET lo que es un estándar ya establecido. - Documentación extensa y completa. - Ligeramente más eficiente que UnityScript y Boo en tiempo de ejecución en Unity.	- Menos fácil de aprender para alguien sin experiencia con lenguajes POO. - Menos ejemplos en la documentación de Unity.

- Encuentra errores en tiempo de compilación.	

Tabla 3: C#, ventajas e inconvenientes

UnityScript	
Ventajas	Inconvenientes
- Tiene una rápida curva de aprendizaje - Documentación extensa y completa de la documentación de Unity.	- Es un lenguaje creado por los integrantes de Unity para Unity y no tiene un uso externo al entorno. - Encuentra errores en tiempo de ejecución.

Tabla 4: UnityScript, ventajas e inconvenientes

Boo	
Ventajas	Inconvenientes
- Tiene una rápida curva de aprendizaje - Fácil de hacer código rápido y limpio. - Encuentra errores en tiempo de compilación.	- Está claramente menos extendido que C# y UnityScript entre la comunidad. -Difícil de encontrar tutoriales y ejemplos.

Tabla 5: Boo, ventajas e inconvenientes

Es común encontrar proyectos en los que hay fragmentos codificados en un lenguaje y otros fragmentos codificados en otro. Esto es posible en Unity pero hay que asegurarse de que se sigue un estricto orden de compilación de los scripts (el orden de compilación de los scripts es algo parametrizable dentro del editor).

4. Antecedentes

Previa a la realización del proyecto se tuvo que investigar acerca de las distintas tecnologías que iban emplearse en el mismo, realizando, para ello, una revisión exhaustiva de las herramientas que dichas tecnologías ofrecían y un estudio acerca de la usabilidad de las mismas y la compatibilidad con el *framework* sobre el que está construido el entorno *Unity3D*.

4.1 Motor de videojuegos utilizado

Se ha elegido *Unity* **[3]** como motor para el desarrollo del proyecto al ser uno de los motores de videojuegos profesionales cuya fama y uso más se ha extendido en los últimos tres años ante competidores como *Unreal Engine o CryEngine*.

Una de las razones por las que *Unity* ha destacado con respecto a otros motores de videojuegos es que ha sabido reunir tanto a empresas profesionales de videojuegos 3D, desarrolladores de juegos para plataformas móviles, así como a desarrolladores independientes o particulares debido a varias características:

- La posibilidad de escribir código en tres lenguajes: C#, *JavaScript* o *BooScript*. Lo que atrajo a programadores de XNA (C#), programadores de Cocos2D (*Javascript*) y otros usuarios acostumbrados a lenguajes como java.

- Creación de juegos en 2D o 3D. Aunque *Unity* era originalmente un motor orientado a los videojuegos en 3D los usuarios que provenían de *Frameworks* como Cocos2D siempre lo utilizaron para realizar juegos en 2D por lo que Unity tratando de complacer a todos sus usuarios fue realizando cada vez más cambios para habilitar la creación de videojuegos 2D de manera más cómoda con componentes específicos para físicas, *colliders*, atlas de *sprites*, etc en 2D.

- Extensiones del editor y creación de plugins. El editor de Unity puede extenderse mediante scripts en los tres lenguajes descritos superiormente heredando de la clase *EditorWindow*. Esto ha permitido a los usuarios de Unity crear extensiones del editor para agilizar o incluso automatizar ciertas tareas repetitivas o que requerían la coordinación con otro componentes del equipo de desarrollo como artistas o diseñadores de niveles. De la misma forma, estas extensiones han permitido la creación de una amplia tienda (Store) de *plugins* que de manera gratuita o de pago permiten ampliar, o incluso sustituir, funcionalidades en Unity. Un ejemplo de ello es el *plugin* NGUI, un *framework* que suple las necesidades de los programadores para crear interfaces de usuario y *HUDs* debido a que lo ofrecido por *Unity* en este apartado resulta ser muy escaso (aunque un nuevo sistema de interfaces ha sido anunciado para la versión 4.6 por parte de *Unity*).

- La comunidad de *Unity*. El motor cuenta con una amplia comunidad que a través de los foros ayuda a los desarrolladores con cualquier problema o duda sobre como implementar o diseñar diferentes componentes de un videojuego. Además de ello *Unity* cuenta con una extensa documentación, así como, de numerosos tutoriales oficiales en *Youtube* que enseñan cómo utilizar algunas de las herramientas que el editor dispone.

Por otra parte *Unity* presenta algunos inconvenientes debidos principalmente a la existencia de dos versiones del motor, la versión Pro y la versión Free.

Como el propio nombre indica, la versión Pro contiene numerosas herramientas y utilidades necesarias en proyectos más ambiciosos, de la misma forma esta versión requiere un pago para poder utilizarse y compilar el juego para PC, MAC o Linux, si además se quiere exportar el juego a otras plataformas como consolas o móviles utilizando herramientas de la versión PRO es necesario adquirir al versión anteriormente mencionada pero también la compra de manera individual de la extensión pertinente para la plataforma objetivo.

Algunas de las herramientas de la versión Pro que no pueden emplearse en la versión Free son:

- **Navmesh**: Compilación de una malla de navegación sobre los diferentes elementos de la escena marcados como navegables. Utilizado para el movimiento de la inteligencia artificial por entornos con otras entidades y obstáculos dinámicos.
- **Efectos de post-procesado a pantalla completa**: Efectos como *motion blur*, corrección de colores, etc.
- **Creación de lightmaps con luces globales o de área**: Se pre calculan las sombras causadas por luces globales o de área estáticas aplicando el resultado sobre las texturas que componen la escena, ahorrando el cálculo de dichas luces en tiempo de ejecución. El uso de *lightmaps* con resto de luces disponibles (*spotlights, point lights*, etc) si está disponible en la versión Free.
- **Texturas 3D**: Uso de texturas 3D utilizando efectos de *shaders* avanzados. Sombras suaves causadas por luces focales o puntuales en tiempo real: Pero si están disponibles sombras con menor resolución causadas por luces direccionales.
- **LOD (Level Of Detail)**: Cuando una escena se hace demasiado grande se aplican técnicas de reducción del nivel de detalle. Estas permiten disponer de varias versiones de una misma malla con diferentes niveles de complejidad (número de vértices, *shaders*, etc) utilizándola de manera adecuada al distancia al jugador o jugadores.
- Y algunas características de uso menos frecuente como filtros de audio, *occlusion culling y profiler* (monitorización) del tiempo de renderización, físicas y código en gráficas.

En la mayoría de proyectos la versión Free es más que suficiente, sobre todo, si se trata de proyectos 2D.

Además de estas dos versiones cabe mencionar que *Unity* pone a disposición de los usuarios una prueba de 30 días de la versión Pro con la que realizar pruebas de rendimiento y comprobar si de verdad interesa la compra de dicha versión.

En este proyecto se usa la versión Pro de *Unity* haciendo uso de la versión de prueba de 30 días, junto con el *plugin* NGUI así como las librerías necesarias para adaptar *Leap Motion* al videojuego. La razón por la que se utiliza la versión Pro de *Unity* es la creación de mallas de navegación para el movimiento de los enemigos (inteligencia artificial) por ellos de manera más profesional y similar a lo que haría una empresa de videojuegos que utilizase el motor.

4.2 Fundamentos del motor Unity

El entorno de *Unity3D* se ha caracterizado desde sus orígenes por su simplicidad a la hora de hacer uso de las distintas herramientas que nos ofrece a los usuarios. Por ello, alguien que no disponga conocimientos de la industria puede, fácilmente, aprender a manejarlo. Sin embargo, pese a su simplicidad, el entorno ofrece a los usuarios más avanzados unas herramientas con una potencia equiparable a la de motores de desarrollo considerados de mayor nivel.

4.2.1 Interfaz

La interfaz de *Unity* consta de diversas ventanas integradas. Cada una se puede desplazar por la interfaz de *Unity* para crear una configuración personalizada a nuestro gusto (*Layout*). Las ventanas principales son la ventana del proyecto, la ventana de la jerarquía, el inspector, la ventana de escena y la de juego.

- La **ventana del proyecto** es como un explorador de archivos en el cual podemos organizar, agregar y modificar todos los elementos o *Assets* que puede llegar a tener nuestra aplicación. Todos los archivos mostrados se corresponden a los de la carpeta llamada *Assets* dentro de la carpeta *root* del proyecto. Se sincronizará a tiempo real con la carpeta del sistema, esto significa que si añadimos algo a la carpeta y volvemos a *Unity*, se actualizará. También podemos arrastrar cualquier objeto dentro de esta ventana y se añadirá al proyecto.

No hay que tener miedo a la hora de añadir elementos a la carpeta *Assets*, ya que no todo lo que se encuentra en esta carpeta será exportado a la hora de compilar. Solo se añadirán al ejecutable los *Assets* que se usen en alguna escena. En esta ventana también dispondremos de un buscador que nos permite buscar y filtrar todos los componentes.

- La **ventana de la jerarquía** es la que nos sirve como herramienta de organización de los elementos de una escena. Es decir, aquí podremos organizar, añadir, eliminar o buscar cualquier elemento ya instanciado. Para instanciar un nuevo elemento solo tenemos que arrastrar un a*sset* desde la ventana del proyecto a esta ventana o a la escena. También se pueden crear algunos objetos genéricos desde el menú, como por ejemplo formas geométricas predefinidas (cubo, esfera, cilindro) o algunos *prefabs* ya

configurados, como un sistema de control en primera persona. Esta ventana también aporta algunas funciones útiles, como por ejemplo centrar la ventana de la escena sobre el objeto seleccionado (haciendo doble *click* izquierdo con el ratón) o arrastrar uno de los objetos creados a la ventana del proyecto, lo que creará un *prefab* con la configuración actual del objeto.

- El **inspector** es una ventana que permite observar qué componentes tiene añadido el objeto seleccionado y modificar sus propiedades. Se puede modificar la apariencia del inspector para los *scripts*. Esto puede resultar muy útil para poder configurar desde el editor y por tanto reutilizarlos para otras tareas similares. Aun así, el inspector predeterminado ya dispone de los elementos más comunes como los elementos primitivos, vectores 2D y 3D, curvas de animación, colores, y más.
- La **ventana de escena** contiene una representación del mundo 3D que conforman todos los *assets* instanciados del videojuego. Esta ventana es muy útil sobre todo para poder hacerte una idea de la composición de todos los elementos así como editarlos, moverlos, rotarlos o escalarlos. Puede extenderse y darle funcionalidad extra, como añadir objetos, cambiar propiedades o dibujar elementos aclaratorios como por ejemplo la dirección en la que se va a mover una plataforma, o crear una línea que dibuje la trayectoria de un enemigo, estos elementos se llaman *gizmos*. Además, esta ventana permite que nos movamos por los elementos la escena con unos controles muy limitados, por supuesto, pero podemos observar un acabado parecido al que se verá al final.
- La **ventana del juego** muestra lo que se verá a la hora de iniciar la escena seleccionada. *Unity* puede ejecutar el juego dentro del editor y será en esta ventana donde simularemos que estamos jugando. En esta ventana se pueden poner distintas resoluciones de pantalla para probar que se vea bien en cualquier resolución. Justo debajo de la barra de menús tenemos el botón de *Play* que ejecutará el juego dentro de esta ventana, y el botón de pausa que detendrá la ejecución en caso de que queramos para a inspeccionar algún elemento. Además de estos dos botones tenemos el botón paso a paso que ejecutará *frame* a *frame* para poder inspeccionar con mayor exactitud.

Unity tiene más ventanas, aunque hemos repasado las más utilizadas. Otras ventanas son la ventana de la consola, que muestra los fallos y te redirige a la línea de código correspondiente, y ventanas de configuración de algún *plugin*.

4.2.2 Escena

Una escena en *Unity* se trata de una composición de *Assets*, *scripts* y eventos que conformarán los niveles de juego. Lo más normal es organizar un proyecto de *Unity* por una escena principal con el menú de juego y después una escena por cada nivel. Esto no solo es conveniente para la organización de un videojuego sino que además tiene ciertas repercusiones.

Cuando en *Unity* se carga una escena, se almacena en memoria todo lo que esa escena vaya a utilizar, de esa forma podemos ir liberando espacio en memoria organizando bien las escenas y los cambios entre ellas. El cambio entre escenas no es un procedimiento suave, aunque se puede ir cargando la escena siguiente mientras el juego sigue ejecutándose. En el momento en el que demos la orden para cambiar de escena se eliminarán todos los objetos de la escena antigua y se crearán los objetos de la nueva escena. Si no hemos cargado los objetos con anterioridad, tendremos que acceder a memoria para cargarlos y esto puede ser un procedimiento costoso.

Por si fuera necesario pasar cierta información entre escenas (datos del usuario, el jugador, puntuaciones, etc...) y no quisiéramos depender de una base de datos como intermediario, existe una instrucción (D*ontDestroyOnLoad*) que le indica a *Unity* que ese objeto no debe destruirse al cambiar de escena. Evidentemente tendremos que tener en cuenta que objetos perdurarán entre escenas para poder configurarlas sin problemas.

Otra manera de intercambiar datos entre escenas es usar la clase P*layerPrefs*. Esta clase se trata de una abstracción de una tabla hash para poder guardar números enteros, de coma flotante, y cadenas de texto en disco. Para ello le asignaremos a cada valor una clave, que será una cadena de texto. Esta clase es persistente por lo que además también podemos usarla para diferentes sesiones de juego. También existen *plugins* que permiten guardar objetos enteros en *PlayerPrefs*.

4.2.3 El GameObject y sus componentes

Un *GameObject* **[4]** o instancia de objeto de juego es la unidad básica que conforma *Unity*. Un *GameObject* va a almacenar una serie de componentes que dotarán de comportamiento a este objeto. Cada *GameObject* tiene una variable *enabled* que modula si esta activo o no, cuando este desactivado todos sus componentes dejarán de ejecutar ciertos métodos. Cada componente por separado tendrá también un *booleano* para indicar si esta activo o no, por si queremos desactivar solo parte de la funcionalidad de este objeto. Los métodos que no se ejecutarán cuando el *GameObject* o el componente este desactivado son los heredados de la clase *MonoBehaviour Awake*, *Start, Update, FixedUpdate* y *OnGUI*.

Los *GameObject* siguen una organización jerárquica en forma de padre/hijo, propia de los grafos de escena, que servirá entre otras cosas para organizar los elementos de las escenas. Como el comportamiento de los *GameObjects* depende de sus componentes se vuelven objetos muy versátiles. Además se pueden añadir/obtener componentes en tiempo de ejecución de un *GameObject* para añadir funcionalidades o modificar los valores de las variables para alterar su comportamiento.

Un componente es básicamente un *script* que tiene que heredar de la clase M*onoBehaviour*. *Unity* ya tiene diversos componentes implementados, pero lo más común es que la inmensa mayoría de los *scripts* que utilicemos sean nuestros. Algunos de los más importantes son:

- **Transform:** Esta componente contendrá la posición, rotación y escala de objeto. Todo *GameObject* tendrá siempre un componente *Transform*. Esta componente también tendrá una variable llamada *parent* que referencia a su padre directo en la jerarquía. Si esta componente es nula este objeto no tiene padre y la posición mostrada será la posición global (respecto al punto 0,0,0 de la escena) pero si el padre es otro *GameObject* la posición mostrada será la posición local (con respecto a la posición del padre). También nos ofrecerá métodos para mover y trasladar objetos, rotarlos con respecto a un punto o con respecto a su propio pivote, entre otros.
- **Collider:** Se trata de un componente que va a crear un área de inclusión (caja, esfera, capsula, …). Esto permitirá que el objeto pueda colisionar con otros objetos y que se disparen ciertos eventos en los *scripts* de ese *GameObject*. Este componente tiene un *booleano* para indicar si actúa como un *Trigger*, lo que significa que no bloqueará el paso a otros *colliders* pero sí que disparará los eventos pertinentes.
- **Rigidbody:** Este componente va a permitir que el *GameObject* interaccione con las físicas. Añadir este componente permitirá que la gravedad le afecte y permitirá también añadir fuerzas con una magnitud, un sentido y un tipo de fuerza. Para algunos objetos puede resultar más fácil (o necesario) realizar movimientos mediante su componente *Transform* pero aun así quizá queramos que use el motor físico en algún momento. Para ello deberíamos activar el *booleano* I*sKinematic* que limitará el cálculo del motor físico y podremos mover el objeto cambiando su posición en el *Transform* de manera segura, si no lo hiciéramos de esta manera al cambiar su posición el motor físico lo tomaría como un tele transporte y si se produjera alguna colisión, la fuerza de repulsión sería infinita y por tanto las reacciones serían imprevisibles.

4.2.4 MonoBehaviour

Es la clase principal **[5]** que cualquier *script* que desee formar parte de un *GameObject* debe de heredar. Esta clase nos aportará métodos y eventos útiles para la inicialización, ejecución y destrucción. Algunos de los más importantes son:

- **Start:** Se llama al inicio de la creación del *GameObject* que lo contiene. Este método se usa principalmente para dotar de valores por defecto al s*cript*. A no ser que alteremos el orden de ejecución de los *scripts,* no podemos controlar en qué orden se ejecutarán.
- **Awake:** Este método se ejecutará justo antes del S*tart*. Esta iniciación en dos fases es muy útil ya que puede que algunos *scripts* necesiten para su propia inicialización que otros *scripts* ya tengan sus valores calculados. Y aunque podamos alterar el orden de ejecución de los *scripts* siempre será una solución más práctica usar el *Awake*.

- **Update:** Se trata de un método que se va a ejecutar una vez por cada *frame*, por tanto tenemos que tener mucho cuidado al introducir bucles y operaciones costosas dentro de este método. Sin embargo, es muy útil para realizar movimientos de personajes y/o control del *input*.
- **LateUpdate:** Este método se ejecuta justo después del U*pdate*. Esto puede ser muy útil para que algún objeto reaccione ante otro y siempre reaccione después. Por ejemplo si tenemos un *script* que mueve un personaje y queremos que la cámara le siga, el *GameObject* que contiene la cámara deberá tener un *script* que implemente el *LateUpdate* y aquí modificaríamos la posición de la cámara.
- **FixedUpdate:** Este método se llama justo después de cada iteración del bucle de físicas. Debemos tener en cuenta que no podemos sobrecargar mucho este método ya que esto podría repercutir en reacciones inesperadas dentro del juego.

Heredar de M*onoBehaviour* nos aporta también muchos otros métodos que se accionarán en respuesta a diferentes interacciones con el *GameObject* asociado. Algunos de estos métodos se llaman desde otros componentes del mismo *GameObject* y dependen de ellos para su ejecución. Los más importantes son:

- **OnEnable/OnDisable**: Este método se ejecuta siempre que se ejecute el método *SetActive* con el valor true/false sobre el *GameObject* asociado. Este método puede servir para preparar una interfaz o para guardar objetos siempre que se desactive el objeto.
- **OnDestroy:** Siempre que un *GameObject* se destruya (llamando a la función Destroy o DestroyImmediate) se ejecutará este método en todos sus componentes, lo cual permitirá guardar los progresos o avisar de que se va a destruir a otro *GameObject*.
- **OnMouseDown/Enter/Exit:** Estos métodos ayudan a la hora de capturar cuando el ratón pasa por un *Collider* asociado a ese *GameObject*. Esto permite capturar respectivamente pulsaciones de ratón y cuando entra o sale el ratón del *Collider* asociado.
- **OnCollisionEnter/Stay/Exit:** Este conjunto de métodos se dispararán si el *GameObject* dispone de un *Collider* que no esté en modo *Trigger* y se produce una colisión con otro *Collider* con las mismas características. Reciben un parámetro del tipo *Collision* que nos ofrece toda la información necesaria acerca de la colisión entre dos Colliders.
- **OnTriggerEnter/Stay/Exit:** Funcionan igual que los anteriores pero solo cuando un *Collider* que no sea *Trigger* colisiona/entra/sale de otro que sí lo sea.

4.2.5 Prefabs

Prefab es la contracción de prefabricado. Se trata de un *GameObject* que ya está configurado y que puede replicarse con la configuración de inicio. Junto con el *Prefab* también se guarda la jerarquía que depende de este *GameObject*, de ese modo cuando lo instanciemos también crearemos a todos sus hijos. Gracias a todo esto podemos crear objetos complejos de manera rápida y sencilla, sin tener que estar añadiendo componentes en tiempo de ejecución. Además, si tenemos más de una instancia creada

a partir de un *Prefab* y cambiamos algún componente o variable del *Prefab* cambiará en todas las instancias.

4.3 Audio en Unity3D

Para que una comunicación auditiva funcione correctamente es necesaria la presencia de al menos dos nodos, el emisor y el receptor. Para ello, el entorno de Unity proporciona, por defecto, dos componentes que el desarrollador debe emplear para construir el sistema de audio en un videojuego.

Estos componentes son los siguientes:

- **AudioListener [7]:** Simboliza al receptor de la comunicación y su único papel es el de recibir el audio. De manera habitual se suele colocar este componente en el personaje del jugador, o en la propia cámara.
- **AudioSource [6]:** Este componente, de un nivel de complejidad un poco más elevado que el anterior, corresponde al emisor del audio en la comunicación y proporciona una serie de funciones (*Play*, *Stop*, *Pause*, etc) que deben llamarse desde el código por el desarrollador. Además, en este componente pueden configurarse la siguiente lista de parámetros desde el editor:
  - *Volume:* El volumen (0..1)
  - *Pitch:* La velocidad a la que la pista va a reproducirse (-3..3)
  - *Mute:* Si se desea que originalmente el componente esté silenciado
  - *Loop:* Si se desea una reproducción en bucle por defecto
  - *Priority:* La prioridad de este componente respecto a otros *AudioSources*
  - *Min Distance:* Distancia mínima necesaria respecto al *AudioListener* para la percepción del audio
  - *Max Distance:* Distancia máxima permitida respecto al *AudioListener*
  - *Doppler Level:* Nivel al que se desee que actúe el efecto *Doppler*
  - *Volume Rolloff:*Tipo de atenuación del sonido dada la distancia (lineal, logarítmico, etc)

Además, *Unity PRO* proporciona al usuario una colección formada por seis filtros de sonido que son empleados para dotar al audio de un efecto añadido. Estos filtros se resumen brevemente en la siguiente lista:

- **AudioChorusFilter:** Se crea un efecto de coro como si hubiera múltiples de *AudioSources* reproduciendo la misma pista con un *delay* entre ellas.
- **AudioDistortionFilter:** Produce un efecto de distorsión.
- **AudioEchoFilter:** Este filtro simula un echo efectuado tras un *delay* previamente definido.
- **AudioHighPassFilter:** Se deja pasar las frecuencias elevadas y se produce una atenuación de las frecuencias inferiores a la definida en el parámetro *CutOff Frequency*.
- **AudioLowPassFilter:** Se deja pasar las frecuencias bajas y se produce una atenuación de las frecuencias superiores a la definida en el parámetro *CutOff Frequency*.

- **AudioReverbFilter:** Se crea una distorsión con el fin de crear una simulación de un entorno personalizado.

En la siguiente imagen podemos observar el aspecto visual de los distintos filtros cuando se colocan como componente a un *GameObject* de la escena.

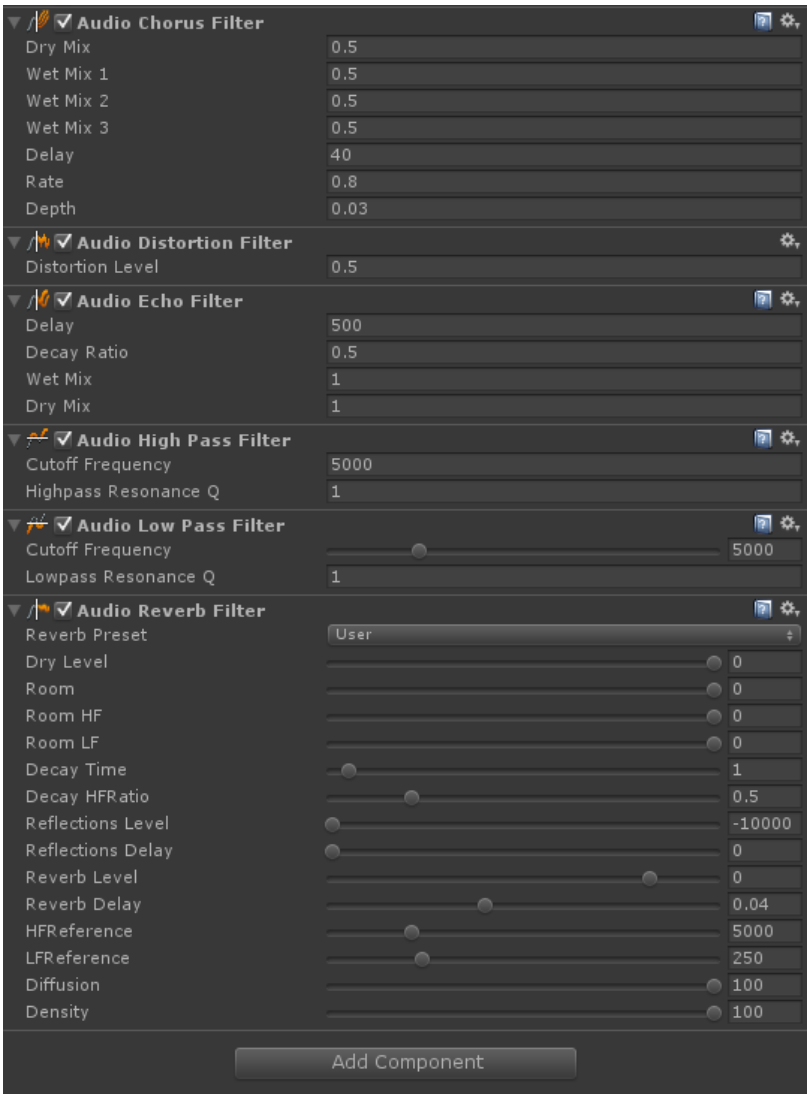

Imagen 1: Filtros de audio

4.4 Accord .NET

Este *framework* permite a los desarrolladores realizar aplicaciones de ámbito científico en *.NET* por medio de una colección de librerías construidas sobre el *framework AForge.NET*. Esta plataforma se subdivide en una colección de librerías que abarcan desde modelos de redes neuronales, vectores soporte o modelos ocultos de *Markov* a Tratamiento de imágenes o audio. Esta librería **[8]** puede encontrarse en las versiones 3.5 y 4.0 del *framework .NET*.

La plataforma se divide en las siguientes librerías:

- **Accord.Math**: Librería auxiliar para el uso complejo de matrices o algoritmos de optimización numérica.
- **Accord.Statistic**: Esta librería contiene distribuciones probabilísticas, modelos ocultos de Markov y modelos de análisis discriminantes.
- **Accord.MachineLearning**: Esta librería contiene máquinas de vectores soporte, árboles de decisión modelos de *Bayes*, modelos de mixturas Gaussianas y una serie de algoritmos como pueden ser el RANSAC o el de validación cruzada.
- **Accord.Neuro**: Librería basada en redes neuronales.
- **Accord.Imaging:** Librería para la realización de transformaciones en imágenes.
- **Accord.Audio:** Esta librería ofrece una documentación para el procesamiento, transformación y filtrado de señales de audio.
- **Accord.Vision:** Librería empleada para el reconocimiento y tratamiento facial en tiempo real.
- **Accord.Controls:** Histogramas y visualizadores de datos para su uso en aplicaciones científicas.

Uno de los objetivos de este proyecto eran la realización de un sistema de reconocimiento de trazos con el que el jugador puede crear patrones y luego emplearlos para controlar cualquier aspecto del juego. Para ello se ha empleado la librería *Accord.Math* y *Accord.Statistic* ya que estas permiten la creación de modelos ocultos de *Markov* que es la opción que se escogió inicialmente para poder crear esta arquitectura.

4.5 Leap Motion

Este dispositivo fue desarrollado con el fin de llevar la interacción con el ordenador a un nuevo nivel. El dispositivo, de 0.5 pulgadas de alto, 1.2 de ancho y 3 de longitud, ofrece al usuario un campo de interacción con la forma de una pirámide truncada invertida con un campo de visión de 150 grados en el que las manos o herramientas (lápices, punteros, ...) se detectan con una precisión de 1/100 milímetros. Además, aparte de disponer de una precisión extraordinaria, el dispositivo realiza su computo logrando una tasa de 200 *frames* por segundo.

Una peculiaridad de este aparato es que realiza las lecturas en un espacio tridimensional (x,y,z) lo que ha permitido elaborar aplicaciones en las que las manos tienen un papel imprescindible, esto puede apreciarse en la siguiente imagen.

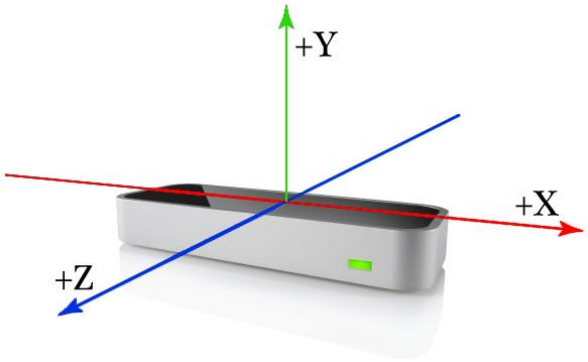

Imagen 2: Coordenadas en Leap Motion

Para poder emplear el *LeapMotion* se ha dado soporte a distintos lenguajes de programación (*JavaScript*, *C# y Unity, C++, Java, Python y Objective C*) que se pueden consultar en la página web **[9]** donde además se puede encontrar diversas aplicaciones ejemplo **[10]** y distintas documentaciones actualizadas día a día para sus respectivo lenguaje. Además, hay toda una colección de ejemplos que un desarrollador con una cuenta ya creada se puede descargar con el fin de visualizar la estructura del proyecto, así como todo el código empleado en su ejecución.

Para su uso en *Scares For Sale* y la necesidad de emplear *Unity* en el proyecto se han seguido los pasos de instalación listados en la web en la sección *GettingStarter* **[11]**. En nuestro caso se escogió la opción *Unity / C#*.

Una vez se tiene instalado el *SDK* y se ha realizado el despliegue en el entorno pertinente se puede comenzar a usar el dispositivo. Para ello, se ha de conocer cómo funciona el flujo temporal del mismo y como aprovechar las funciones facilitadas por la librería para leer la información registrada por el aparato.

El *LeapMotion* detecta, en cada ciclo, la posición de las manos y herramientas localizadas en el interior del campo de visión previamente nombrado. Para ello es necesario disponer del acceso a una variable de tipo *LeapMotionController* por medio de la cual podremos acceder en cada ciclo a la información recogida a través de la llamada al método *Frame* (a esta función se le puede pasar por argumento un número de tipo entero, con esto logramos acceder a un *frame* en particular incluido dentro del historial). Esta función nos devuelve un objeto tipo *Frame* que incorpora información acerca de la posición, rotación y escala de las manos y herramientas detectadas. A su vez estos objetos *Hand* recopilan la información de todos los dedos detectados para esa mano **[12]**.

Por defecto la documentación nos ofrece la gestión de cuatro gestos que deben inicializarse en el *Leap Motion Controller* para indicar el deseo de que el dispositivo realice la lectura de los mismos. En la inicialización se pueden indicar valores para los distintos parámetros de configuración con el fin de sustituir a los que vienen por defecto.

- **Swipe:** Cuando se realiza un movimiento de barrido con la mano en una dirección., tal y como se representa en la siguiente figura.

Imagen 3: Swipe

- **ScreenTap:** Cuando el usuario incide con el dedo en dirección a la pantalla como está ilustrado en la siguiente imagen.

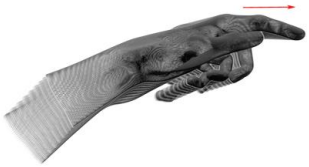

Imagen 4: ScreenTap

- **KeyTap:** Cuando se realiza un movimiento en *zigzag* en vertical con el dedo siguiendo el patrón establecido en la siguiente figura.

Imagen 5: KeyTap

- **Circle:** Cuando se traza una trayectoria circular con el dedo con un radio determinado como puede observarse a continuación.

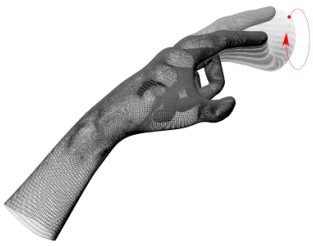

Imagen 6: Circle

5. Análisis y diseño

En una etapa previa a la implementación de los distintos puntos del proyecto se realizó una fase de diseño en el que se analizaron los problemas a resolver así como las distintas formas de solventarlos para obtener una solución lo más eficiente y simple posible, con el fin de poder emplearla en posibles proyectos de videojuegos futuros.

Las soluciones planteadas se han ideado con el objetivo de que cualquier usuario ajeno al sistema sea capaz de utilizarlo empleando para ello un tiempo mínimo de adaptación, dado que cuando se realiza el montaje del videojuego se desea minimizar los gastos temporales en aprendizaje del entorno.

5.1 Sistema de entrada

La aparición de los *smartphone* de nueva generación (*iPhone , Apple 2007*) propició que millones de personas en todo el mundo comenzaran a apreciar la industria del videojuego de una forma en la que anteriormente con las videoconsolas no había podido ser posible. Estos jugadores denominados en la industria como *casual gamers* hicieron posible la aparición de un nuevo mercado en la que las tecnologías móviles y los videojuegos iban establecer una alianza de alta rentabilidad.

Con el fin de crear una experiencia única para el usuario se hizo uso de la tecnología táctil, ya disponible en los primeros *smartphones*.
Esta tecnología ofrecía al usuario la capacidad de controlar el transcurso del juego directamente por medio de sus dedos o de una herramienta óptica como podía ser un puntero óptico.
Además, estos dispositivos han sido dotados de giroscopios, acelerómetros, micrófonos y demás herramientas y sensores que han sido usados en la industria para incrementar la usabilidad de estas aplicaciones.

Indistintamente del desarrollo para plataformas móviles, videoconsolas o ordenador, *Unity* aporta a los desarrolladores un abanico muy amplio de posibilidades a la hora de controlar el sistema de entrada para los videojuegos. A todas las funcionalidades posibles se puede acceder a través de la librería *Input* brevemente resumida en la siguiente lista de conceptos.

- **Control del ratón:** Acceder dentro de un *frame* (fotograma) a la posición del ratón y si se ha pulsado o se ha soltado el botón derecho, el izquierdo o la rueda del mismo.
- **Control de touches:** Detectar la entrada de uno o más dedos en la pantalla del teléfono así como sus fases: *began, moved* y *ended*.
- **Eventos de teclado y controladores de videoconsolas:** Conocer si se ha pulsado o se ha soltado alguna tecla contenida en el teclado o en el mando.
- **Brújula:** Por medio de esta funcionalidad se puede conocer la dirección respecto a los puntos cardinales a la que está orientado el terminal.

- **Acelerómetro:** Esta herramienta proporciona al usuario la información relacionada con la inclinación y la velocidad a la que se está moviendo el dispositivo.
- **Micrófono:** Este medio de detección de audio puede emplearse por algunos videojuegos en los que el uso de la voz sea algo requerido para la jugabilidad.

A la hora de diseñar el sistema se escogieron las siguientes mecánicas compatibles tanto para su uso en ordenador como para su uso en dispositivos móviles. Estas funcionalidades son las más usadas a la hora de realizar un videojuego para móviles.

- **Swipe:** Esta forma de entrada se popularizó con la entrada de la tecnología táctil en los videojuegos ya que permitía controlar diversas acciones de una forma muy cómoda e intuitiva para el usuario mediante la acción de realizar un barrido con el dedo o con el cursor del ratón en la pantalla. Esta mecánica se ideó para que fuera totalmente configurable por una serie de parámetros.
- **Pulsación:** Esta mecánica es de las más empleadas en el ámbito de los videojuegos para móviles y consiste en detectar cuando el usuario ha pulsado la pantalla con el dedo o con el ratón y conocer además la posición definida por un vector bidimensional donde se ha realizado esta acción.
- **Selección de objetivo:** Es algo frecuente en muchos proyectos el permitir al usuario que realice una pulsación exclusivamente en los objetos preparados para ello. En el caso de *Scares For Sale* esta función se aplicará en las trampas de forma que el usuario únicamente podrá activar mediante una pulsación aquellos objetos preparados para ello. Sin embargo, aunque Unity permite de una manera indirecta realizar esta funcionalidad en nuestro caso ha sido implementada dentro del sistema por medio del componente *ClickableObjec* el cual se asignará a los objetos que se desee resulten afectados. Además, este sistema nos permite crear la compatibilidad necesaria para su uso con *LeapMotion*.
- **Sistema de eventos:** Gracias a este sistema le ofrecemos al usuario una herramienta de gran potencia que puede emplear para activar desde el editor determinadas funciones cuando un objeto de los previamente nombrados se pulsa. De esta forma permitimos que el usuario ahorre tiempo de programación y se propicia un código menos extenso y de mayor simpleza. Para su uso, el usuario dispone de una lista en cada componente *ClickableObject* preparada para editarse desde el editor de *Unity* donde cada elemento de la lista recibe como parámetros un *GameObject* y el nombre del método a elegir por medio de una lista desplegable.

Para ofrecer estas mecánicas y teniendo en cuenta que se iba a incorporar el uso del *LeapMotion* se decidió implementar una arquitectura modular formada por un controlador central *InputManager* y dos módulos. De esta forma permitimos al usuario configurar todos los parámetros necesarios desde el *InputManager* a través del editor, así como escoger si se desea hacer uso del *LeapMotion* o no de forma que en tiempo de ejecución se realizará la inicialización de los módulos necesarios. En la siguiente imagen se puede apreciar la relación unitaria que existe entre los módulos realizados.

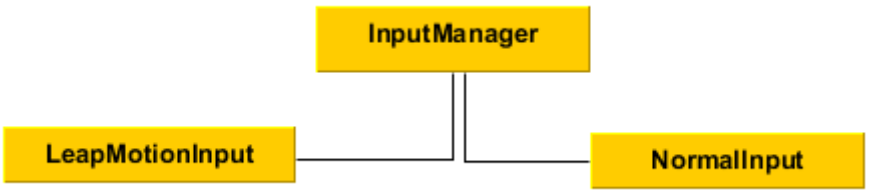

Imagen 7: Modelo uml arquitectura del sistema de entrada

Como se ha mencionado previamente *InputManager* actúa como un manejador central de forma que todos los eventos (*OnClick, OnSwipe, …*) tienen su fuente en él. Con esta estructura permitimos gran flexibilidad a la hora de incorporar nuevas mecánicas y se facilita la realización de nuevas extensiones en un futuro. Este controlador también será el encargado de comprobar cuando se ha detectado una pulsación por parte del usuario si esta se ha realizado sobre un objeto dotado del componente *clickableObject*.

5.2 Reconocedor de trazos

Conforme avanzan las tecnologías de reconocimiento de formas se ha creado una gran gama de videojuegos que hacen uso de estas para crear una experiencia de juego totalmente nueva. Un claro ejemplo se encuentra en el videojuego *Brain Training del Dr. Kawashima ¿Cuántos años tiene tu cerebro?* **[13]**, publicado en todo el mundo entre el 2005 y 2006 para la consola *Nintendo DS*. Este juego ofrecía al jugador un abanico de mini juegos mentales, algunos de los cuales empleaban el reconocimiento de voz o el reconocimiento de trazos dentro de sus funcionalidades.

Sin embargo, si un desarrollador desea hacer uso de estas mecánicas para un juego en *Unity* deberá recurrir a la tienda (Unity *Asset Store* **[14]**) y aún así encontrar un *plugin* que te ofrezca una implementación para ello es una tarea ardua y difícil, donde, por supuesto los resultados encontrados serán de pago. Por ellos se tomó la decisión de añadir un nuevo módulo al sistema de entrada que permitiría al usuario crear y detectar patrones mediante trazos en una forma ergonómica y sencilla de usar.

El primer problema a plantear fue la toma de decisión del modelo a emplear (redes neuronales, máquinas de vectores soporte, modelos ocultos de *Markov*, etc). Finalmente la decisión tomada fue la de emplear modelos ocultos de *Markov* dado su uso extendido en el campo del reconocimiento de trazos.

Una vez se había escogido el modelo a usar se realizó una investigación acerca de las librerías con licencia *GNU* que abarcaran este campo y pudieran emplearse en el sistema sin necesidad de realizar una implementación completa del modelo. Finalmente se encontró la librería *Accord.NET* ampliamente usada en investigaciones y ámbitos docentes que además ofrecía una versión de la misma para *.NET* 3.5, la versión sobre la que está construido el motor de *Unity* de modo que la compatibilidad era, en teoría, la necesaria. Sin embargo, en unos primeros intentos de test se comprobó que a pesar de ser la versión adecuada, el *engine* respondió demostrando una serie de incompatibilidades con ciertas librerías contenidas en el total de *Accord*. Este problema pudo solventarse por medio de la adquisición de un fragmento del total

cuyo contenido eran todas las librerías necesarias para la utilización de los modelos ocultos de *Markov* **[15]**.

Una vez se habían solucionado los problemas de compatibilidad había que pensar cómo hacer uso de ella. Para ellos se optó por realizar una capa de abstracción donde el usuario solo debería preocuparse de cargar la información y guardarla mediante una llamada a los métodos contenidos en esta capa y de usar cuando lo requiera la función *Evaluate* pasándole como parámetros una lista de puntos. Como se muestra en la siguiente imagen, el componente *GesturePatternManager* es el que almacena la referencia a todos los patrones creados.

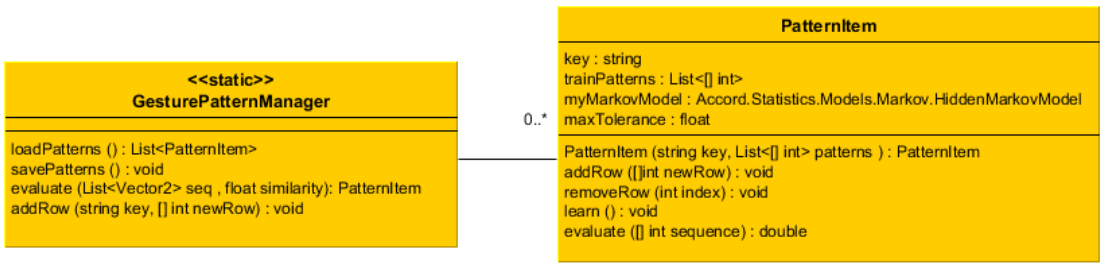

Imagen 8: Modelo uml reconocedor de trazos

Para la creación de los patrones se decidió realizar un subeditor dentro de *Unity* destinado para tal fin en el que el usuario pudiera crearlos y entrenarlos de una forma cómoda y sencilla. Pero la información creada a partir de este editor es temporal, por ello se diseñó un sistema de ficheros que sería el utilizado para mantener la persistencia de estos datos. El sistema en cuestión se formó empleando un archivo a modo de índice en el que todas las claves de los patrones están registradas y una sucesión de archivos, uno por cada patrón, en los que se almacenan todas las secuencias numéricas a emplear en el entrenamiento de cada uno de trazos.

Cuando se desea tratar con las secuencias de puntos hay dos grandes problemas a los que se tuvo que hacer frente.

- Por un lado surgió la necesidad de limitar el número de puntos a emplear dado su coste en memoria, ya que un alto coste suponía tiempos de carga innecesarios a la hora de trabajar con el sistema.
- Por otra parte, en unos primeros estudios con los modelos ofrecidos por la librería, se comprobó que la similitud obtenida a la hora de evaluar una secuencia en los patrones disminuía considerablemente conforme se incrementaba la cantidad de puntos recogidos. Tras una fase de análisis, se demostró la hipótesis de que esto se debía a la falta de normalización en las secuencias recogidas y a su dependencia en los *frames* por segundo a los que el editor trabajaba en el momento de la recolección.

Para solucionar estos problemas se planteó una solución canónica para los datos. Esta consistía en dividir la lista recogida en *N* trazos de forma que solo fueran empleados los puntos situados en los extremos de cada trazo. Con esto se consiguió reducir el número de recursos a utilizar en un factor de *N*/longitud(secuencia total), lo que propició una mejora en los tiempos altamente apreciable. Además, con estos trazos evitábamos la posible degeneración de la secuencia por el número de *FPS*. La separación de las

secuencias en trazos se realiza de forma proporcional como se puede apreciar en la siguiente figura.

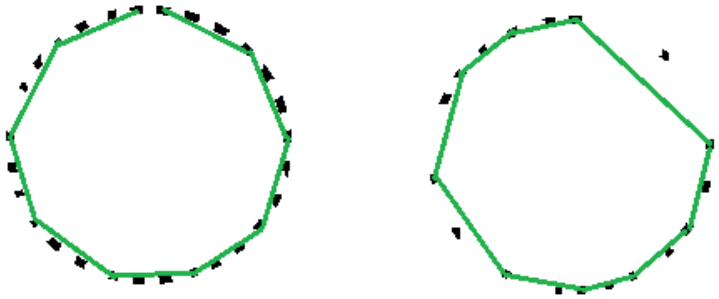

Imagen 9: Separar secuencias de puntos en N trazos

Una vez se disponía de la secuencia ya separada solo restaba codificarla para su almacenamiento en los ficheros correspondientes. Para ello, el proceso a realizar es codificar los trazos obtenidos en función al sector al que su vector dirección pertenezca. Es decir, para cada uno de los trazos se le asigna un índice numérico en función a la pendiente de su vector dirección. Esto se refleja en la imagen mostrada a continuación.

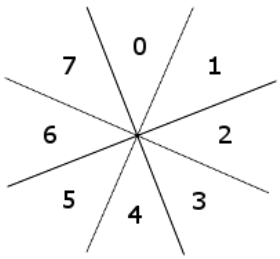

Imagen 10: Codificación de las secuencias

5.3 Sistema de audio

El audio es un componente fundamental en un videojuego aunque a veces este adquiere un papel secundario y se prioriza la elaboración de la jugabilidad o de los demás aspectos del juego. Esto ocurre en muchas ocasiones por la falta de una arquitectura robusta y fácil de utilizar que permita dotar de música y demás sonidos a la aplicación. Para corregir esto se decidió realizar la implementación de este sistema con el fin de emplearse no solo en *Scares For Sale* sino también en proyectos futuros, empleando para tal fin la *API* ofrecida por *Unity* con cierta ayuda de librerías encontradas en el *framework .NET* 3.5.

El sistema ideado se puede considerar como una capa de abstracción al ya presente dentro del editor. Sin embargo, con esta arquitectura se pretende no solo agilizar el proceso sino también acercarlo al usuario, haciendo más fácil su usabilidad mediante la implementación de nuevas mecánicas que ofrezcan soltura en la implementación,

reduciendo la cantidad de código necesario y permitiendo una mayor limpieza del mismo.

La estructura elegida comienza con la función del *AudioManager*. Este controlador, siguiendo el patrón *Singleton*, es donde el desarrollador puede crear los canales que van a estar disponibles a la hora de reproducir los distintos sonidos y regular una función definida entre 0 y 1 que será la que se utilice en la regulación del volumen de los canales. Una vez creado un canal el usuario podrá configurarlo totalmente desde el editor cambiando, entre otros factores, la prioridad de cada canal.

Por otro lado, la otra parte del sistema corresponde al componente *SoundEmitterObject*. Este script, que puede asignarse a cualquier *GameObject* de la escena, representa un foco fuente de sonido. Para ello se le debe asignar un canal de los creados en el manejador central por medio de la clave identificativa correspondiente al deseado. Esto provoca la suscripción por parte del componente al canal asignado de forma que se hará uso de la configuración definida por él canal.

Además, el componente *SoundEmitterObject* se ideó para que cada uno de las pistas de audio asignadas pudieran configurarse desde el editor indicando su clave (la utilizada para seleccionarlo), si su ejecución es en bucle, la distancia mínima y máxima a la que el sonido puede escucharse por el receptor, el intervalo temporal que se emplea de cada pista (inicio y fin) y por último una lista de eventos similar a la implementada para el sistema de entrada anteriormente citado donde para cada uno de los eventos se define el punto temporal de la pista en la que se lanzará, el objetivo de tal evento y el método que se desea ejecutar. Este método podrá, al igual que en el sistema de entrada, escogerse mediante una lista desplegable donde se muestran aquellos métodos de nivel de protección pública cuya lista de argumentos sea nula y su valor de retorno sea tipo *void*.

Por último se decidió dotar al sistema de respuesta visual al usuario durante el tiempo de ejecución de forma que el usuario pueda contemplar el rango de alcance mínimo y máximo del audio cuando se esté emitiendo. Además, el usuario podrá comprobar si un *SoundEmitterObject* tiene un *clip* en ejecución mediante la visualización de una textura que indicará el estado actual (en esta versión en vez de emplear una textura se utiliza un *gizmo* de una esfera, verde si se está reproduciendo un sonido y rojo en caso contrario), tal y como aparece en la siguiente imagen.

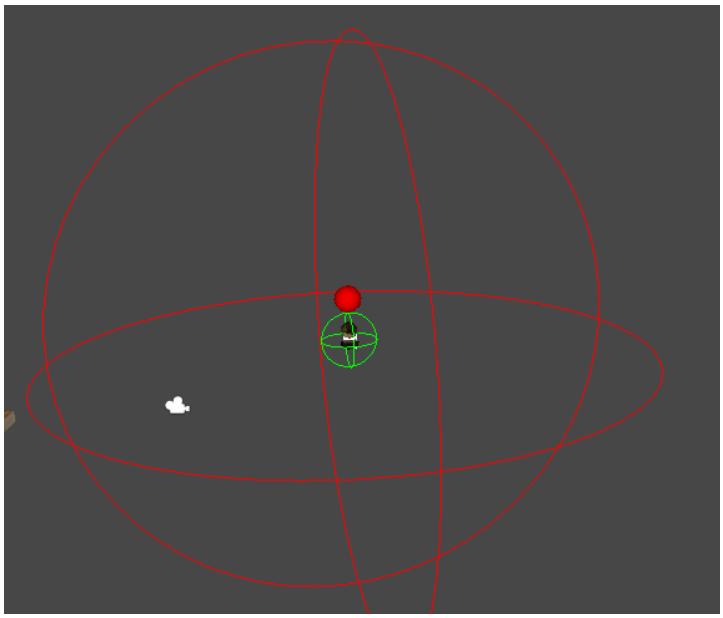

Imagen 11: Respuesta visual del sistema de audio

5.4 Cámara y *alpha* de los materiales

Durante la realización del videojuego *Scares For Sale* surgió la necesidad de realizar un script que se encargara de gestionar el movimiento de la cámara durante el *gameplay* así como de las animaciones *fade in* y *fade out* del componente visual de los elementos de la escena que pudieran suponer dificultades visuales para el espectador.

A la hora de construir los materiales para todos los objetos de la escena se ha debido tener en cuenta que hay ciertos *GameObjects* a los que se deberá realizar una animación de su canal *alpha*. Por ello, a estos se les asignaron los *shaders* **[16]** siguientes contenidos en el grupo *Transparent Shader Family*.

- **Transparent Diffuse:** Para aquellos objetos de los que el artista solo facilitó la textura. Este se ha empleado en objetos que la cámara no detectará con un gran detalle cómo puede ser el caso de las armas empuñadas por las armaduras o los objetos menudos empleados a modo de ambientación. En la siguiente imagen se aprecia como al material del candelabro se le ha asignado un *shader* difuso con una textura.

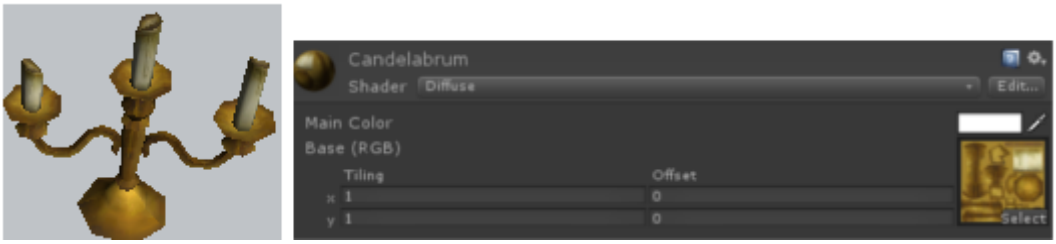

Imagen 12: Shader transparent diffuse

- **Transparent Bumped Diffuse:** Para aquellos a los que el artista además añadió un mapa de normales. Sus casos de uso han sido los objetos de mayor relevancia que guardan alguna relación con los sustos o los objetos de mayor

tamaño los cuales la cámara si puede visualizar con un mayor detalle, además de su aplicación obvia en los posibles compradores de la mansión. A continuación se muestra el material con *shader transparent bumped diffuse* aplicado a las librerías.

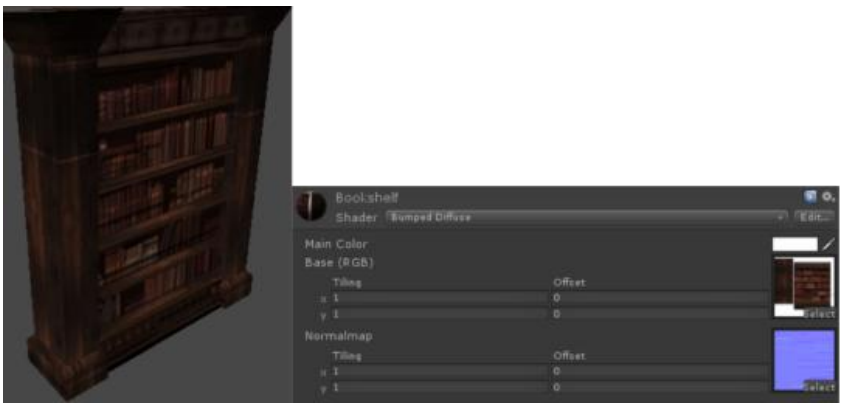

Imagen 13: Shader transparent bumped diffuse

- **Transparent Parallax Diffuse:** Hay cierta cantidad de objetos a los que se le añadió un extra mediante el uso de un mapa de alturas para realizar, por medio de este *shader* una simulación más avanzada de la profundidad. Su uso ha sido empleado especialmente en los materiales relacionados con el edificio (paredes, chimenea) y algunos objetos de gran tamaño. Este *shader* se ha aplicado a las paredes de la mansión, tal y como se aprecia en la imagen siguiente.

Imagen 14: Shader transparent parallax diffuse

Para poder realizar el movimiento de la cámara se decidió desde el comienzo que la mansión tendría un armazón cuya base estaría formada por un cuadrado de lados iguales. De esta forma posibilitamos que el movimiento se realice mediante una trayectoria circular con centro localizado en el punto central de la base.

Por otro lado, puesto que la mansión está formada por diferentes pisos y se realiza una transición desde el tercer piso a la azotea pasando por un tramo de tejado, se decidió implementar una forma basada en *offsets* que pudiera usarse para definir la distancia a recorrer durante el movimiento vertical de la cámara cuando el usuario decide realizar un cambio de planta.

6. Implementación

Una vez se dispuso de una base con el diseño ya creado, se procedió a la implementación del código necesario de los distintos sistemas planteados para el proyecto, empleando en los casos necesarios la ayuda de la documentación de las librerías pertinentes.

6.1 Sistema de entrada

Para la codificación del sistema se comenzó con la realización del manejador central usado como intermediario entre los distintos módulos mencionados anteriormente, cuyas variables pueden observarse en la siguiente tabla.

Nombre	**Tipo**	**Descripción**
Instance	InputManager	Usada para acceder al manager mediante el patrón singleton.
clickableObjectLayer	string	Layer empleada para los objectos con el componente *ClickableObject*. Debe estar creada en el editor.
Swipe	enum	Empleado para indicar el tipo de swipe detectado.
enableLeapMotion	bool	Si el valor lógico es *true* al comienzo de la ejecución, se desplegará el módulo *LeapMotioInput*.
customCursorImage	Texture2D	La imagen asociada se empleará como cursor cuando se habilite el *LeapMotion*.
enablePatternRecognition	bool	Si al inicio de la ejecución tiene un valor lógico *true* se habilitará el uso del reconocedor de gestos.
patternRecognitionKey	KeyCode	Usada para habilitar la detección de trazos (*NormalInput*).
minSwipeDuration	float	Tiempo mínimo de recorrido de un swipe (*NormalInput*).
maxSwipeDuration	float	Tiempo máximo de duración de un swipe (*NormalInput*).
minSwipeDistance	float	Recorrido mínimo en pixeles (*NormalInput*).
maxSwipeDistance	float	Recorrido máximo en pixeles (*NormalInput*).
minSwipeSpeed	float	Velocidad mínima del swipe en pixeles / segundo

		(*NormalInput*).
parameterConfig	GestureConfiguration	Instancia con la información de inicialización de los parámetros del *LeapMotion*.

Tabla 6: Parámetros del componente InputManager

Este manejador, denominado *InputManager*, es el encargado de inicializar los distintos módulos durante la inicialización mediante el método *Awake* de la clase *MonoBehaviour*. En esta fase, se realizan las distintas acciones.

- Se le asigna el valor de la instancia actual a la variable *Instance* usando para ello la palabra reservada *this*.
- Se añade el componente *NormalInput* que será el encargado de gestionar las distintas mecánicas para ordenador y plataformas móviles.
- Si la variable enableLeapMotion tiene un valor lógico *true* se añade el componente *LeapMotionInput* y se inicializa mediante la llamada a la función *initialize* pasando como argumento la variable *parameterConfig* que contiene la configuración necesaria.
- Si la variable *enablePatternRecognition* está marcada como *true* se cargan los patrones guardados en el sistema de ficheros y se realiza su entrenamiento.

En este componente se contienen los eventos empleados como salida del sistema. Estos eventos son los siguientes.

- **onSwipe():** Este evento se lanzará en el momento que se haya reconocido el gesto pertinente por parte de cualquiera de los módulos.
- **onClick():** Este evento se lanza cuando se produce la detección de una pulsación por parte del usuario. Sin embargo, previo al lanzamiento del evento se comprueba si esta pulsación ha sido sobre un objeto marcado con la *layer* previamente creada *ClickableObject*.

  Para realizar la detección de estos objetos se ha hecho uso de la herramienta *Raycast* **[17]** contenida en la librería de físicas de *Unity*. Esta utilidad funciona lanzando un rayo desde un origen empleando un vector dirección y una distancia máxima de forma que si se produce una colisión con el componente *Collider* de un *GameObject* se almacena esta información en una variable de salida de tipo *RaycastHit* **[18]**. Como argumento adicional se puede emplear una máscara construida mediante el desplazamiento de N bits, donde N es el número asignado a la *layer* deseada, para provocar que todo objeto que no tenga asignada la *layer* pertinente sea ignorado por el rayo. De esta forma aseguramos que solo se obtendrá información en la variable de salida en el caso de producirse una colisión con los objetos deseados. A continuación se muestra la realización de los rayos desde la cámara aplicando en su ejecución una máscara para ignorar todo aquel objeto que no sea *Clickable*.

```
private ClickableObject DoClick(Vector2 mousePosition)
{
    //We generate the mask of the ClickableObject layer
    int layerMask = 1<<LayerMask.NameToLayer(clickableObjectLayer);
    RaycastHit hit;
    Ray ray = Camera.main.ScreenPointToRay(mousePosition);  //This
    if (Physics.Raycast(ray, out hit,Mathf.Infinity,layerMask))
    {
        if (hit.transform != null) {
            return hit.transform.GetComponent<ClickableObject>();
        }
    }
    return null;
}
```

Imagen 15: Detección de ClickableObjects

En el momento que se ha detectado la selección de un *GameObject* con el componente *ClickableObject* se realiza una llamada al método *executeAction* que sobrescribirse en caso de querer ampliar la funcionalidad de este script. Cuando se invoca esta función se realiza el llamamiento a los distintos eventos construidos en el componente que vienen recogidos en una lista.

Para la implementación de estos eventos se ha implementado mediante una clase adicional denominada *OnClickEvent*. Todos los parámetros de esta clase son mostrados mediante una extensión del editor de forma que el usuario puede definir, para todos los elementos de la lista de eventos, el *GameObject* objetivo y el método a invocar mediante el uso de una lista desplegable donde se muestran solo aquellos métodos que no requieran de argumentos y no devuelvan ningún parámetro. Cuando el usuario escoge el nombre del método por medio de la lista desplegable este se almacena en una variable interna de forma que en el momento de invocar al evento se utiliza este nombre para buscar en el objetivo el objeto de tipo *MethodInfo* **[19]** correspondiente a la función. Una vez encontrado se realiza la llamada a la función *Invoke* pasándole como argumentos el objetivo de la llamada y un *array* de longitud cero.

Además, la clase *ClickableObject* realiza una función extra. En la fase de inicialización se utiliza la variable definida en el manejador central (*clickableObjectLayer*) para hacer que al *GameObject* se le asigne la *layer* a emplear a la hora de realizar los *raycast* por el método anteriormente citado.

Puede parecer algo innecesario el uso de esta clase ya que *Unity* por defecto proporciona un método denominado *OnMouse* que es llamado cuando se produce una pulsación sobre él. Sin embargo, la necesidad de emplear esta clase se debe a que se desea emplear el *LeapMotion* que, por desgracia, no proporciona una integración total con la *API* del *engine* y no se realiza la detección por defecto de estas colisiones cuando se detecta desde el *LeapMotionSDK* que el usuario ha introducido un cuerpo en la zona de toque.

Con el componente *InputManager* y la clase *ClickableObject* ya definidas se prosiguió con la codificación del módulo *NormalInput*. En este *script* se realiza, por un lado el control de las pulsaciones, por otro el del *swipe* y por último el de la recogida de secuencias en el caso de tener el reconocedor de patrones activo. Es decir, este *script* se encarga de la gestión básica del *Input* sin la intervención del dispositivo *LeapMotion*.

Puesto que tanto para una pulsación como para un *swipe* se produce una fase inicial (*Clicked*) y una fase final (*NonClicked*), se utilizó una variable de tipo *enum* para distinguir estas fase dentro del flujo de juego en el método *Update* de *MonoBehaviour*. De esta forma cuando el proceso se encuentra en estado *NonClicked* y el usuario pulsa el ratón o pulsa mediante un dedo en la pantalla se realiza una transición al estado *Clicked*. Mientras se esté en la fase *Clicked* se realiza un conteo del tiempo mediante la suma en cada *frame* de la variable *Time.deltaTime* (esta variable corresponde al tiempo pasado desde el anterior *frame* y el actual). Una vez el usuario levanta el dedo de la pantalla o del ratón se comprueba si el tiempo ocurrido, la distancia recorrida y la velocidad empleada son lo suficientemente extensos como para considerarse un *swipe*, en caso contrario se estaría hablando de un toque. Además, en caso de exceder el tiempo máximo establecido se cancela la acción produciéndose un retorno a la fase *NonClicked*.

Cuando el usuario mantiene pulsada la tecla asignada para el reconocimiento de trazos se hace uso de la fase *Clicked* para realizar la recogida de los puntos donde el usuario sitúa el cursor o el dedo. Una vez acaba esta fase el sistema procesa la secuencia, la codifica y devuelve el patrón más similar de entre todos los entrenados. En caso de soltar la tecla el reconocimiento se cancela y se realiza un *reset* de la lista secuencia obtenida a espera de comenzar una nueva.
Para evitar problemas de precisión en dispositivos móviles en la versión actual del sistema se deshabilita durante la inicialización el reconocimiento de varios *Touch* **[20]** por parte de *Unity*. Esto se debe a que cuando hay varios dedos en pantalla, el *engine* sitúa la posición realizando una media entre las de todos los dedos reconocidos.

A continuación se realizó la codificación del módulo *LeapMotionInput* encargado de recoger, interpretar y proporcionar la información necesaria para poder realizar el control del sistema mediante las manos sin necesidad de un periférico de entrada.

En primer lugar se implementó la fase de inicialización que se ejecuta, en el momento de crearse el componente por el manejador central, mediante la función *initialize*. Esta función recoge como argumento un objeto de tipo *GestureConfiguration* en cuyo interior está almacenada la información necesaria para configurar el módulo.

Nombre	**Tipo**	**Descripción**
swipeEnabled	bool	Booleano empleado para habilitar el reconocimiento de swipes por parte del SDK.
leapSwipeMinLength	float	Longitud mínima necesaria que se debe recorrer con la mano para detectar un swipe (mm).
leapSwipeMinVelocity	float	Velocidad mínima necesaria para que el SDK reconozca un swipe (mm/s).
circleEnabled	bool	Booleano empleado para habilitar el reconocimiento del gesto Circle.

leapCircleMinRadius	float	Radio mínimo que debe tener el circulo en el recorrido realizado por la mano (mm).
leapCircleMinArc	float	Arco mínimo del circulo a realizar antes de detectarse (radianes).
screenTapEnabled	bool	Booleano empleado para habilitar el reconocimiento de este gesto.
leapScreenTapForwardVel	float	Velocidad mínima en el eje Z necesaria para reconocer el gesto (mm/s).
leapScreenTapHistorySeconds	float	Tiempo mínimo a emplear para realizar el gesto (segundos).
leapScreenTapMinDistance	float	Distancia mínima a realizar antes de detectarse (mm).
keyTapEnabled	bool	Booleano empleado para habilitar el reconocimiento de este gesto.
leapKeyTapForwardVel	float	Velocidad mínima en el eje Z necesaria para reconocer el gesto (mm/s).
leapKeyTapHistorySeconds	float	Tiempo mínimo a emplear para realizar el gesto (segundos).
leapKeyTapMinDistance	float	Distancia mínima a realizar antes de detectarse (mm).

Tabla 7: Variables de la clase GestureConfiguration

Con el fin de controlar cada uno de los posibles gestos habilitados en la fase anterior se creó una clase base *GestureHandler* con una funcionalidad básica en común para los tres. A esta clase se le dotó de una función *UpdateData* encargada de gestionar el transcurso y control del tiempo para cada uno de los gestos. Además, la clase cuenta con un evento que recibe un argumento genérico que es lanzado en el momento de detectar el gesto pudiendo así emplear un argumento de distinto tipo para cada uno en particular. Es decir, cuando se reconocen *KeyTap, ScreenTap y Circle* el argumento a enviar es la posición actual del *Pointable* (dedo, herramienta, ...) que haya sido el causante del mismo. Mientras que para el *swipe* es una variable de tipo *InputManager.SwipeType* (*Up, Down, Left, Right*) y para *CustomGestureHandler* un *objeto PatternItem* correspondiente al patrón detectado por el reconocedor.

Sin embargo, para ofrecer una mayor escalabilidad en el futuro, se recurrió a la creación de las clases *CircleHandler, KeyTapHandler, SwipeHandler, ScreeenTapHandler y CustomGestureHandler*. Estas clases extienden la funcionalidad

de la clase *GestureHandler* y sobrescriben los métodos antes mencionados con el objetivo de que cada una pueda gestionar su información de manera independiente y así poder, en un futuro, ampliar la funcionalidad para dotar a estos controladores de una mayor precisión y usabilidad.

Con las instancias de estos *handlers* creados y recogidos en una lista empleando el polimorfismo ya es posible realizar la siguiente fase. Esta consiste en controlar, con ayuda del *LeapMotionSKD* **[12]**, la aparición de estos gestos. Para ello y usando la variable *leapMotionController* de tipo *Leap.Controller* podemos acceder a la información referente a la detección de entidades dentro de la zona sensorial de dispositivo por medio del objeto de tipo *Frame* accesible a través de *leapController.Frame()*. Este objeto, además de la información referente a las entidades, almacena una lista (*frame.Gestures()*) de objetos *Leap.Gesture* que corresponde a los distintos gestos reconocidos en ese *frame*. En función del tipo de cada uno se traspasa la gestión al controlador correspondiente de los anteriormente creados que se encargará de evitar la ejecución consecutiva en un intervalo corto de tiempo, ya que un gesto aparece en la lista a lo largo de todo su recorrido siempre y cuando se cumpla con los requisitos de la fase de inicialización.

En cuanto al reconocimiento de los patrones, en caso de estar habilitado, se realiza por medio del controlador *CustomGestureHandler*. En su interior, en la función *UpdateData*, se detecta cuando una entidad traspasa un umbral en el eje *Z (*TouchZone*)*, de modo que en el intervalo de tiempo que dicha entidad se encuentre más allá de ese umbral se irán recogiendo puntos de pantalla como su equivalente en *NormalInput*. En el momento que el usuario regrese a una profundidad "normal" se cortará la cadena y se enviará el evento con el *PatternItem* correspondiente al patrón reconocido.

Otra de las funcionalidades a implementar en el diseño original del sistema era el uso de un cursor personalizado dado que al emplear el dispositivo el usuario no tiene forma visual de saber a qué localización de la pantalla está apuntando en un momento dado. Para ello se diseñó originalmente un modelo en el que el usuario debería realizar una calibración en función de la distancia *offset* relativa del dispositivo a la pantalla. Este modelo presentó el inconveniente de que el proceso de calibración se convertía en algo necesario a realizar en una fase previa al uso del *LeapMotion* de modo que la idea fue desechada y tras un proceso de investigación en la *API* se encontró la solución con el uso de la clase *InteractionBox*.

Esta clase define un *viewport* constituido por una caja interior a la zona de visión del dispositivo que permite obtener dada una posición en milímetros unas coordenadas normalizadas (0.0 .. 1.0). Con estas coordenadas se convierte en un problema trivial convertirlas a coordenadas de pantalla por medio del producto de las coordenadas normalizadas *X,Y* por las dimensiones de la pantalla en pixeles *width, height*.

En cuanto al cursor, se crea un objeto *GUITexture* **[21]** durante la inicialización del *LeapMotionInput* y se le asocia la textura escogida desde el editor para tal fin. Tras su creación se hizo uso de la función *Update* para actualizar la posición de la textura aplicándole un *offset* en todo momento correspondiente a las coordenadas convertidas obtenidas del *Pointable* (dedo, herramienta, ...) de mayor profundidad en el eje *Z*.

6.2 Reconocedor de trazos

Inicialmente y con el fin de ofrecer al usuario una abstracción de los patrones se decidió implementar un editor dentro del propio entorno en el que se pudiera realizar la creación y el entrenamiento de los mismos. En este editor se muestra un campo de texto donde escribir la clave del patrón y una ventana vacía donde dibujar con el ratón los distintos trazos para su entrenamiento. De modo que si se realiza un dibujo de algún trazo y se pulsa el botón *Add!!!* se usará para el entrenamiento del patrón con la clave asociada o se creará si no existiera en ese momento. Por otro lado, mediante el botón *Save!!!* ejecutamos la acción de guardado del componente *PatternItemManager* por el cual toda la información se almacena, garantizando así la persistencia.

Esta ventana, codificada como una extensión de la clase *EditorWindow* **[22]** de las librerías de *Unity*, no ejecuta la función *Update* de modo que no se ha podido emplear para la recolección de los puntos para durante la realización del trazo. En su lugar se tuvo que indicar a la ventana que recogiera los eventos de movimiento de ratón asignando un valor *true* a la variable *wantsMouseMove* de la instancia de la ventana. A continuación, desde la función *OnGUI* se realizó la lectura de los eventos registrados en cada *frame* mediante la variable estática *current* de la clase *Event* **[23]**. Gracias a esto se podía detectar cuando el usuario pulsaba o levantaba los botones del ratón así como los movimientos del mismo a través del tipo de evento recogido en *type* en el interior de *current* (MouseDown, MouseUp, MouseMoved, ...).

En cuanto al *renderizado* del dibujo realizado con el ratón se ha empleado como ayuda una clase estática *Drawing* **[24]** obtenida en los foros de la comunidad de *Unity*. Esta clase cuenta con una función *DrawLine* que, dados dos puntos, dibuja una línea de dos dimensiones uniéndolos. De modo que si para cada par de puntos consecutivos de la secuencia recogida realizamos una llamada a esta función pasándolos como argumentos obtendremos el dibujo del trazo tal y como ha sido realizado por el usuario.

Sin embargo los datos obtenidos del editor no son nada si no realizamos las transformaciones pertinentes en la clase *PatternItemManager* por medio de la función *Serialize()* que realiza la codificación de la colección de puntos de entrada en una lista de enteros (0 .. 7) de tamaño *N* donde *N* es el número de trazos empleados para separar.

Esta clase, además, es la encargada de almacenar la información acerca de todos los patrones creados mediante una lista de *PatternItems*. Esta clase está compuesta por una clave a modo de identificación del patrón, un modelo oculto de *Markov* (*Accord.Statistics.Models.Markov.HiddenMarkovModel*) **[25]** y una lista de trazos, ya codificados, empleados para su entrenamiento. De forma que en el momento de añadir un nuevo trazo a la colección se producirá el entrenamiento del modelo para mantenerlo actualizado en todo momento.

Una vez se dispone de los patrones ya entrenados se puede realizar una llamada a la función estática *Evaluate()* pasándole como argumento la colección de puntos recopilados durante la escritura del trazo. Esta función transforma la cadena en una lista a tratar por el modelo mediante la llamada a *Serialize()* para, a continuación, realizar una evaluación para cada uno de los posibles patrones. Esta evaluación se realiza mediante la llamada a la función propia de los objetos *PatternItem* (*Evaluate()*)

y devuelve la similitud con el patrón evaluado. De esta forma podemos realizar una comparación y devolver aquel con una mayor similitud como el patrón reconocido. Cabe destacar que siempre que el usuario recoja una cadena de puntos y evalúe dicha colección obtendrá un patrón como resultado.

Por último, para mantener la persistencia de toda la información se optó por crear un sistema de ficheros. La jerarquía del mismo se compone de un archivo *keys.txt* y una sucesión de archivos <*name*>.*txt* donde <*name*> es el identificador del patrón correspondiente. El guardado de los datos en estos ficheros se ha realizado por medio de las clases *System.IO.File* **[26]** y *System.IO.Directory* **[27]** del *Framework .NET*. y se almacena en un directorio llamado *GesturePatternInfo* ubicado en el *path* definido por la variable *Application.dataPath* **[28]**.

6.3 Sistema de audio

El cuerpo del sistema de audio reside en el uso del script AudioManager, encargado de la creación de los canales de audio y su configuración, y la colaboración que este realiza en ejecución con los componentes *SoundEmitter*, encargados de gestionar las pistas de audio de un objeto determinado dentro de la escena de *gameplay*.

En primer lugar se realizó la implementación del *manager*. Este componente sigue el patrón *singleton* por medio de una variable estática *Instance* cuya inicialización se realiza al iniciar la escena de juego usando el valor proporcionado por la palabra reservada *this*. La función principal de este *script* es la de almacenar los distintos canales de audio creados por el desarrollador a través del editor. Para ello se realizó una extensión del editor de *Unity* para facilitar la creación, modificación y eliminación de estos canales. Este *manager*, además, añade al *GameObject* asociado el componente *AudioListener* que corresponde al receptor del audio de la aplicación, ya que en la escena solo puede haber un único objeto con este componente.

Dichos canales (*Channel*) se identifican por medio de una clave entera y cuentan con un evento *OnChannelModified* que se lanza cuando se realiza una modificación de los distintos parámetros de configuración, llamando de este modo a las funciones *onChannelInfoChanged* de los objetos *SoundEmitter* de la escena que se hayan suscrito a dicho canal por medio de la función *Suscribe*. De este modo mantenemos estos componentes actualizados. Otro parámetro muy importante de la clase *Channel* es la variable *entera priority* (valores 0 .. 255) utilizada para la atenuación del volumen, efecto que puede configurarse desde el *manager* por medio de una *AnimationCurve* **[29]** cuya edición se realiza visualmente a través del editor. En la versión inicial del sistema realizado para este proyecto se implemento la gestión estática del volumen. Es decir, cuando un *SoundEmitter* lanza una pista de audio el componente *AudioSource* asociado adquiere el valor de volumen correspondiente al canal al que está suscrito. Este valor (0.0 .. 1.0) se obtiene mediante la llamada a la función *Evaluate* de la *AnimationCurve* pasándole por argumento la prioridad del canal normalizada en base uno (*priority* / 255).

Una vez implementada la parte de gestión de los canales se procedió a implementar el componente *SoundEmitter* cuya labor es la causante de la reproducción del audio de toda la aplicación. Cuando se le asocia este *script* a un objeto de la escena, hay que

indicarle desde el editor el número correspondiente al identificador del canal al que se va a asociar. Con este parámetro, en la llamada a la función *Awake*, se realiza una llamada a la función *Suscribe* del canal correspondiente. A continuación, se añade el componente *AudioSource* empleado posteriormente como reproductor de los *AudioClip* asociados.

En cuanto a la gestión de las pistas de audio se realizó por medio de una lista de objetos *AudioObject* cuya edición se realiza desde el entorno de *Unity* por medio de una extensión del editor. De este modo podemos añadir, modificar o eliminar los *items* de esta lista de una forma ergonómica. Estos objetos están compuestos por las siguientes variables:

Nombre	**Tipo**	**Descripción**
name	string	Nombre (ID) del clip
clip	AudioClip	Archivo de audio
loop	Bool	Si se desea reproducir en bucle
startPoint	float	Punto en el que iniciar el clip
endPoint	float	Punto en el que finalizar el clip
minDistance	float	Distancia mínima auditiva
maxDistance	float	Distancia máxima auditiva
audioEventList	List <AudioEvent>	Lista de eventos

Tabla 8: Variables de la clase SoundEmitter

Los eventos almacenados en la lista *audioEventList* son objetos *AudioEvent,* similares a los eventos empleados en el sistema de entrada. Estos realizan una llamada a una función prefijada desde el editor en el momento en el que el clip del *AudioObject* en el que se encuentran alcanza el tiempo definido por una variable denominada *time*. Esta variable solo puede adquirir el rango de valores (startPoint .. endPoint).

Por otro lado, la realización de la información visual proporcionada por el componente *SoundObject* durante el tiempo de ejecución, se ha realizado por medio de la función *OnDrawGizmos* de *MonoBehaviour*. Dentro de este método se dibujan dos esferas de radios *minDistance* y *maxDistance* mediante la función *Gizmos.DrawWireSphere* y una esfera situada sobre el objeto de color verde si se está produciendo una reproducción o roja en caso contrario por medio de la llamada a *Gizmos.DrawSphere* **[30]**.

6.4 Cámara y *alpha* de los materiales

Por último, una vez codificados los distintos sistemas a emplear dentro del juego *Scares For Sale* se produjo la realización del movimiento de la cámara, necesario para la transición del jugador a lo largo de las distintas plantas desde distintos puntos de vista. Como se ha mencionado en la fase de diseño, el movimiento se realiza mediante una colección de parámetros denominados *offsets* de forma que para el movimiento entre cada par de plantas se puede decidir el valor numérico del desplazamiento a realizar.

El comportamiento aquí descrito de la cámara se ha realizado por medio del *script CameraController* que, al igual que otros *managers* pertenecientes a los otros sistemas, sigue el modelo del patrón *singleton* por medio de una variable estática. Este componente da la posibilidad al usuario de configurar el comportamiento de la cámara mediante los siguientes parámetros.

Nombre	Tipo	Descripción
numberOfFloors	int	Número de plantas
numberOfWalls	int	Número de caras de la casa
currentFloor	int	Piso actual
currentWall	int	Pared actual
floorsOffset	List<float>	Lista de desplazamientos entre pisos
floors	List<Floor>	Elementos de cada piso y para cada pared que han de cambiar su canal alpha durante los movimientos
timeToCompleteRotation	float	Duración de la rotación entre paredes
timeToCompleteVerticalMovement	float	Duración del desplazamiento entre pisos
timeToCompleteAlphaAnimation	float	Tiempo en realizar la transición del canal alpha (0.0-1.0, 1.0-0.0)

Tabla 9: Parámetros de CameraController

Para empezar se realizó el movimiento entre las distintas plantas del edificio. Para ello, se implemento un método denominado *OnSwipe* suscrito al evento *onSwipe* del *manager* del sistema de entrada, de forma que se pudiera detectar cuando el usuario había realizado un *swipe* y de este modo actuar en consecuencia. Dentro de esta función se comprueba el valor del argumento (*InputManager.SwipeType*) proporcionado por el evento y se inician los movimientos hacia arriba, hacia abajo, rotar hacia la izquierda o rotar hacia la derecha en función de dicho valor, siempre que no haya un movimiento en marcha con el que pudiera entrar en conflicto. Es decir, no se permite realizar un giro cuando se esté girando y del mismo modo no se permite moverse verticalmente si se está desplazando en ese momento o si ese movimiento se sale de los pisos preestablecidos.

Para la realización del desplazamiento vertical se mantiene un registro del espacio recorrido deteniendo el movimiento una vez se ha alcanzado el destino. Este movimiento se ejecuta de forma continua en cada *frame* dentro de la función *Update*, desplazando cada vez una cantidad calculada por la operación *(space / timeToCompleteVerticalMovement) * Time.deltaTime*. Cuando se inicia un movimiento de este tipo, se inicia a su vez la animación del canal *alpha* de los objetos (paredes) pertinentes (Las paredes del nuevo piso se animan de 1.0-0.0 y las del piso de salida de 0.0-1.0).

En cuanto a la rotación, se ha realizado por medio de la función *RotateAround* aplicada al *Transform* del *GameObject* asociado a la cámara usando como pivote *Vector.Up* localizado en el centro de la mansión. De esta forma se realiza un movimiento circular rodeándola. Esta operación se ejecuta dentro de la función *Update* cuando está activa y realiza un desplazamiento en grados calculado por la operación *(space / timeToCompleteRotation) * Time.deltaTime* **[32]**, de forma que cuando se ha recorrido un número de grados suficiente (*360 / numberOfWalls*) se detiene el movimiento. Al igual que con el desplazamiento entre plantas, se realiza la animación del canal *alpha* de los objetos necesarios.

Dentro de este sistema se tiene en todo momento el conocimiento acerca del piso y la pared hacia la que está mirando el usuario. Estos dos enteros se han empleado como índices para poder acceder a los objetos (paredes, cuadros, ...) cuyo material ha de modificarse a la hora de cambiar el valor del *alpha*. Para ello se ha empleado la lista de objetos *Floor* donde cada objeto a su vez tiene una lista de *Wall* con una serie de objetos de la escena asociados. De este modo, conociendo el piso origen, la pared de origen, el piso destino y la pared destino se puede obtener todos los objetos necesarios. Esta lista puede modificarse desde el editor permitiendo cambios rápidos en ella siempre que el usuario lo requiera tal como se muestra en la siguiente imagen.

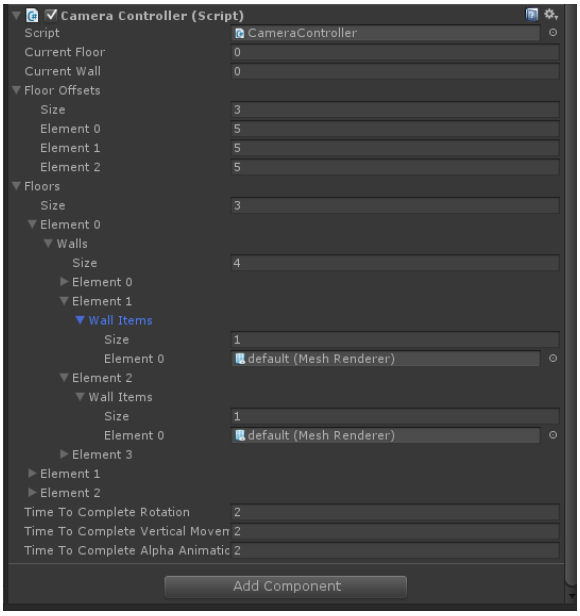

Imagen 16: Componente CameraController

7. Resultados

Una vez finalizada la implementación solo faltaba realizar un último reconocimiento para asegurar que se habían cumplido todos los objetivos marcados para este proyecto.

Para empezar se comprobaron las metas establecidas para el sistema de audio planteado. Es decir, el sistema implementado era capaz de separar por canales de audio todos los sonidos del juego, estableciendo, para cada canal en particular, una configuración distinta. Además, gracias al sistema de prioridades y a la *AnimationCurve* del *manager*, se dotó al sistema de una forma de dispersión del volumen modificable desde el editor. En cuanto a las pistas de audio, no solo se codificó la gestión del intervalo de tiempo a reproducir, sino que además se permitía elegir las distancias a las que dicho sonido podían escucharse. Además, a estas pistas de audio se le añadió un sistema de eventos totalmente configurable desde el editor para la ejecución de funciones en un punto temporal específico de la pista. En la siguiente imagen se muestra el resultado final del componente *SoundEmitter* listo para configurarse.

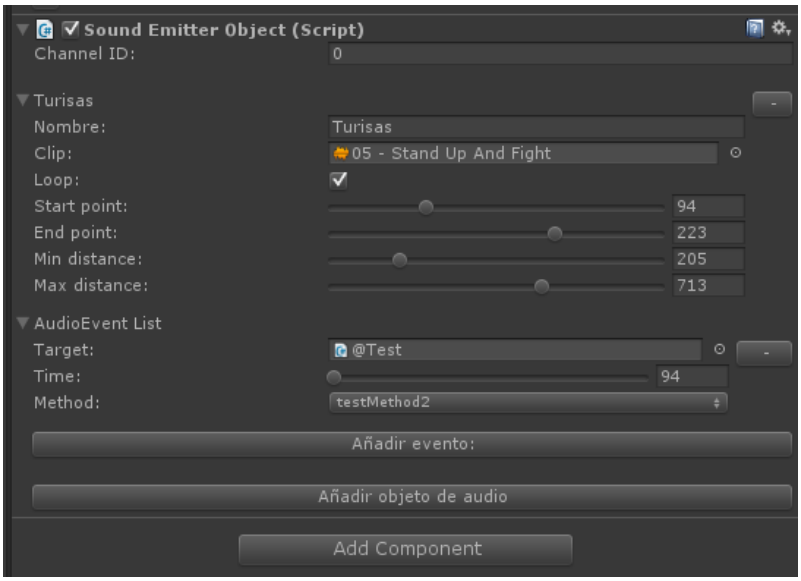

Imagen 17: Componente SoundEmitter

En segundo lugar se realizó la comprobación del sistema de entrada. La implementación realiza la función de caja negra con eventos de salida tal y como fue diseñado. Este sistema gestiona de una forma precisa los distintos eventos de entrada realizados por el usuario (*swipe, click*, ...), tanto en plataformas móviles como en ordenador, mediante el uso de unos parámetros de configuración editables desde el editor con el fin de que el usuario del sistema les asigne el valor más ajustado a sus necesidades. En la siguiente imagen se muestra el componente final preparado para configurarse por el usuario.

Imagen 18: Componente InputManager

Como extra, se dotó a los *ClickableObject* de un sistema de eventos totalmente configurable desde el editor con el que gestionar las llamadas a funciones que se deben realizar en el momento de la pulsación. Esta funcionalidad se muestra en la siguiente imagen, donde como objetivo se asignó el *script* denominado *@Test* de un *GameObject* ajeno al que contiene el *ClickableObject* y como método se seleccionó la función *testMethod* contenida dentro de dicho *script*.

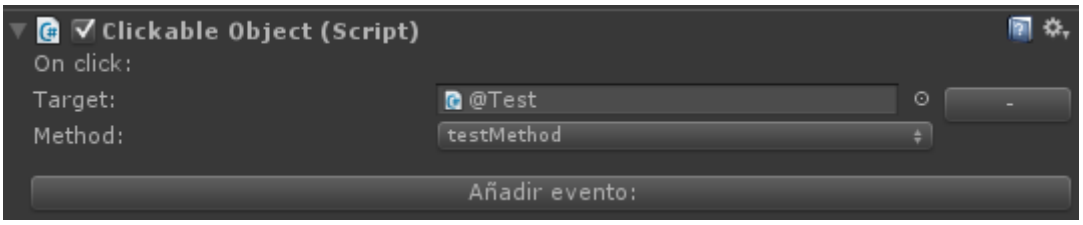

Imagen 19: Component ClickableObject

A continuación se añadió al sistema la posibilidad de emplear el dispositivo *LeapMotion* sin perder las funcionalidades antes descritas y se añadió, como extra, la implementación de un cursor para esta extensión totalmente configurable por el usuario (textura y color). El editor en cuestión es el contenido en la siguiente imagen, en la que además se muestra un ejemplo de reconocimiento del número dos.

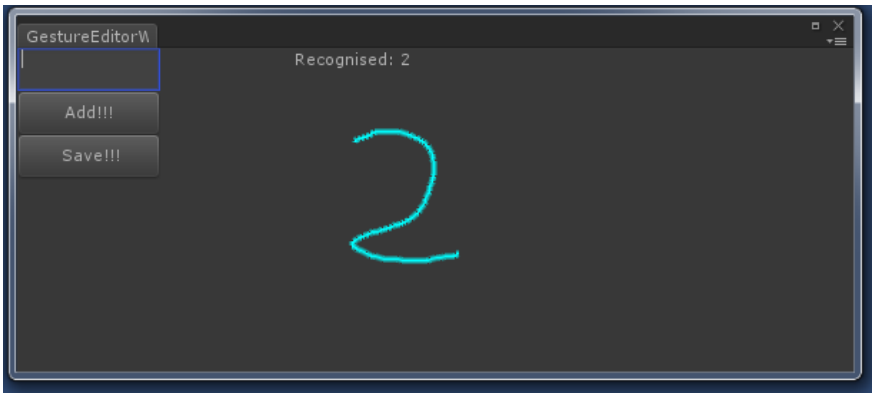

Imagen 20: Editor de patrones

Por último, al sistema de entrada se le dotó de un reconocedor de trazos empleando para ello modelos ocultos de *Markov* encontrados dentro de las librerías *Accord.NET*. En dicho sistema se resolvieron los problemas planteados como la codificación y serialización de las secuencias o su almacenamiento y carga (persistencia). Además, a este reconocedor se le creó un editor dentro de *Unity* en el que se permitía la creación y edición de los patrones a reconocer durante la ejecución y se proveía al usuario de la información visual de los trazos realizados en dicho editor durante la creación de los mismos.

8. Conclusiones y trabajos futuros

Con el trabajo realizado en este proyecto se ha obtenido toda una colección de herramientas que podrán emplearse, en un futuro, en distintos proyectos de desarrollo de videojuegos con el entorno *Unity3D*. Gracias a ellas se agilizará en gran medida el desarrollo de ciertos campos que hasta el momento habían sido implementados, de forma individual, para cada uno de los juegos realizados. De esta forma, el tiempo de desarrollo ahorrado gracias a estas herramientas, nos permitirá dedicar una mayor proporción de tiempo a reforzar otros aspectos del juego.

Puesto que la industria del videojuego cambia continuamente en un futuro se llevará a cabo la realización en de nuevos y más desafiantes escenarios para el juego *Scares For Sale*. En cuanto a los sistemas desarrollados en este proyecto, se realizará una mejora de rendimiento y de mecánicas. Las mejoras pensadas para su realización en un futuro cercano son las siguientes.

- **Reconocedor.** Se ha planteado la remodelación casi completa del sistema de almacenamiento de los patrones en el reconocedor. El cambio implica la sustitución de los ficheros de texto por archivos *JSON* con contenido cifrado.
- **Reconocedor.** Se mejorará el rendimiento del reconocedor en cuanto a la colaboración con el *manager* del *LeapMotion* con el fin de darle estabilidad y evitar posibles bloqueos del programa.
- **Reconocedor.** Se actualizará el editor de patrones permitiendo visualizar y eliminar, dado un patrón, todas las cadenas introducidas para su entrenamiento. Además, se añadirá la opción de distinguir entre *corpus* de entrenamiento y de testeo. De esta forma se podrá comprobar cuál es la fiabilidad de un patrón en un momento dado.
- **Audio.** La gestión del volumen de los canales se modificará para que sea dinámica en función al comienzo y detención de las distintas pistas de audio.
- **Audio.** Se añadirá la gestión de *AudioFilters* personalizada para cada pista de audio.
- **Sistema de entrada (Leap Motion).** Se ampliará el sistema con la nueva versión del *SDK* de forma que se utilicen las nuevas técnicas de *Skeletal Tracking* para incrementar la usabilidad y la precisión del mismo.

9. Bibliografía

[1] UnrealEngine. [Internet] Disponible en: <https://www.unrealengine.com/>

[2] CryEngine. [Internet] Disponible en: <http://cryengine.com/>

[3] Unity3D. [Internet] Disponible en: < http://unity3d.com/es>

[4] Unity documentation (2014). GameObject [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/GameObject.html>

[5] Unity documentation (2014). MonoBehaviour [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

[6] Unity documentation (2014). AudioSource [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/AudioSource.html>

[7] Unity documentation (2014). AudioListener [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/AudioListener.html>

[8] Accord Framework .NET. (2009 - 2014) [Internet] Disponible en: <http://accord-framework.net/>

[9] LeapMotion/DeveloperPortal. [Internet] Disponible en: <https://developer.leapmotion.com/v1>

[10] LeapMotion/Examples. [Internet] Disponible en: <httpshttp://developer.leapmotion.com/gallery>

[11] LeapMotion/GettingStarter. [Internet] Disponible en: <https://developer.leapmotion.com/documentation/csharp/index.html>

[12] LeapMotion/API. [Internet] Disponible en: <https://developer.leapmotion.com/documentation/csharp/api/Leap_Classes.html>

[13] BrainTraining. [Internet] Disponible en: <https://www.nintendo.es/Juegos/Nintendo-DS/Brain-Training-del-Dr-Kawashima-Cuantos-anios-tiene-tu-cerebro--270627.html>

[14] Unity3D/AssetStore. [Internet] Disponible en: <https://www.assetstore.unity3d.com/en/>

[15] Hidden Markov Models in C#. (2010) [Internet] Disponible en: <http://www.codeproject.com/Articles/69647/Hidden-Markov-Models-in-C>

[16] Unity3D/ShaderGuide. [Internet] Disponible en: <http://docs.unity3d.com/Manual/Built-inShaderGuide.html>

[17] Unity documentation (2014). Raycast [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/Physics.Raycast.html>

[18] Unity documentation (2014). RaycastHit [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/RaycastHit.html>

[19] MSDN Library. MethodInfo (Clase) [Internet] Disponible en: <http://msdn.microsoft.com/es-es/library/system.reflection.methodinfo(v=vs.110).aspx>

[20] Unity documentation (2014). Touch [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/Touch.html>

[21] Unity documentation (2014). GUITexture [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/GUITexture.html>

[22] Unity documentation (2014). EditorWindow [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/EditorWindow.html>

[23] Unity documentation (2014). Event [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/Event.html>

[24] Unifycommunity wiki (2013). DrawLine [Internet] Disponible en: <http://wiki.unity3d.com/index.php?title=DrawLine>

[25] Accord.NET Framework (2009 - 2014). HiddenMarkovClassifier Class [Internet] Disponible en: <http://accord-framework.net/docs/html/T_Accord_Statistics_Models_Markov_HiddenMarkovClassifier.htm>

[26] MSDN Library. File (Clase) [Internet] Disponible en: <http://msdn.microsoft.com/es-es/library/system.io.file(v=vs.110).aspx>

[27] MSDN Library. Directory (Clase) [Internet] Disponible en: <http://msdn.microsoft.com/es-es/library/system.io.directory(v=vs.110).aspx>

[28] Unity documentation (2014). Application [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/Application.html>

[29] Unity documentation (2014). AnimationCurve [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/AnimationCurve.html>

[30] Unity documentation (2014). Gizmos [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/Gizmos.html>

[32] Unity documentation (2014). Time [Internet] Disponible en: <httphttp://docs.unity3d.com/ScriptReference/Time.html>